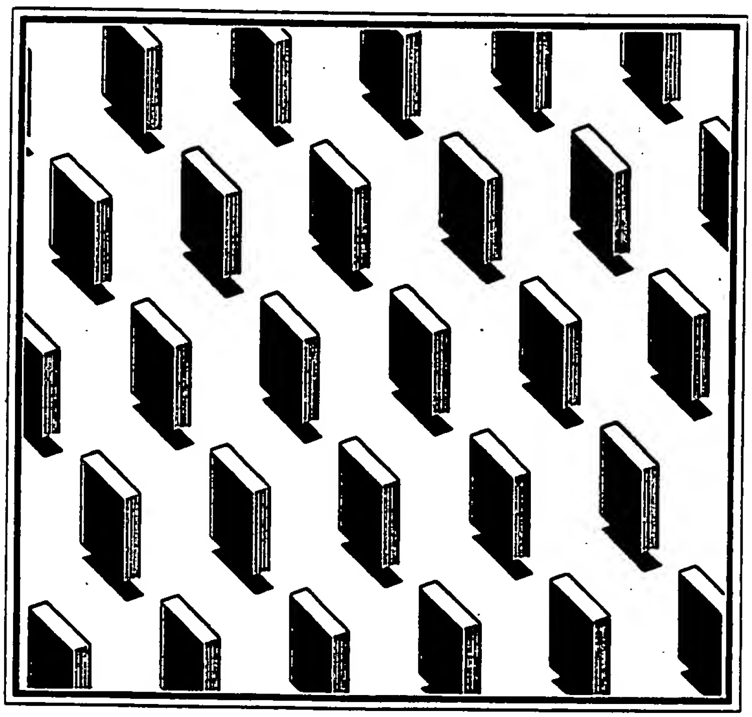
APPENDIX 3

EMPOWER

Reference



for EMPOWER/CS

PERFORMIX

LIMITATION OF LIABILITY

COPYRIGHT

U.S. GOVERNMENT RESTRICTED RIGHTS

Software Version 1.0.1, User's Guide Version 1.0.1

PERFORMIX

PERFORMIX, Inc.
8200 Greensboro Drive, Suite 1475
McLean, VA 22102
(703) 448-6606 (phone)
(703) 893-1939 (fax)

Table of Contents

1.0 Introduction

This manual contains technical reference material for the EMPOWER/CS script functions and EMPOWER/GV functions and commands.

The EMPOWER/CS *Script Development* and *Multi-User Testing* Manuals guided you through the steps required to create and execute sophisticated load testing scripts with EMPOWER/CS. This manual provides reference information to help you understand how to use the various script functions and commands available.

1.1 Organization of the EMPOWER/CS User's Guides

The complete documentation for EMPOWER/CS includes three user's manuals which include general use information, installation instructions, technical reference material, and examples. The following list identifies each user manual:

EMPOWER/CS Script Development
Describes how to create and execute scripts to perform realistic load tests on your client/server environment (This process involves using the EMPOWER/CS tools Capture and Cscc and editing and enhancing your scripts to make them unique to your environment.)

Multi-User Testing
Describes how to use the multi-user tools Mix, Extract, Report, Draw, Monitor, and Global Variables (GV) for emulating realistic loads and measuring performance data

EMPOWER/CS Reference
Describes all commands, functions, and possible error messages for EMPOWER/CS (This manual also includes technical support information for contacting PERFORMIX, Inc.)

1.2 Organization of this Manual

This *Reference* Manual is divided into the following sections:

Section 1:	Introduces you to the EMPOWER/CS reference manual
Section 2:	Describes EMPOWER/CS script functions and GV commands
Section 3:	Lists and describes possible warning and error messages generated by EMPOWER/CS
Section 4:	Provides information on contacting PERFORMIX, Inc. technical support

1.3 User's Guide Conventions

The conventions followed in this User Guide are listed below:

Regular Font	Used for all regular body text
Arial Font	Represents the MS Windows environment
`Mono-Spaced Font`	Used for all command, function, and file names; for all examples; and, generally, for any computer-generated text
`Bold Mono-Spaced Font`	In examples, represents entries made by the EMPOWER/CS user
`[-S scriptid]`	In command syntaxes, text within these square brackets represents optional command parameters
`gv_stat (name | -s)`	Vertical lines (|) separate command parameter choices
`...`	Within scripts, the ellipsis marks indicate that some script content was left out for brevity

`Endfunction()`	Parentheses are included with script functions mentioned in regular body text. For most functions, one or more parameters will be listed in the parentheses.
`Beginscenario()`	EMPOWER/CS script functions use initial capitalization
`gv_stat`	EMPOWER/CS command names use all lowercase letters
Capture	When an EMPOWER/CS tool is mentioned within regular body text, it is shown in regular font with initial caps

The term, "SUT," refers to your client/server system under test. The client/server SUT includes the database server and the database applications on that server used for your emulation.

[This page intentionally left blank]

2.0 Reference

This section provides technical descriptions for EMPOWER/CS script functions and EMPOWER/GV functions and commands. All function names begin with an uppercase letter and command names begin with a lowercase letter.

Note: While Global Variable functions are added to a script .c file and can be executed only when the script is compiled with Cscc, Global Variable commands are entered at the shell prompt of the UNIX driver machine and also may be used during script execution with the Mix tool.

Technical descriptions are presented in the following format:

FUNCTION or COMMAND	The name of the function or command
SYNTAX	The syntax of the function or command
DESCRIPTION	A definition of each parameter (if applicable) and a discussion of the features of the function or command
RETURN VALUE (if applicable)	The value of the return code for successful and unsuccessful execution of the function or command
EXAMPLES	One or more examples showing the use of the function or command in the script
SEE ALSO (if applicable)	A list of related functions or commands

```
Fioreadline("info");
if (FIOLEN==-1){
Fiorewind("info");
Fioreadline("info");
}
Type("%s", FIOBUFFER);
```

SEE ALSO Fioautorewind, Fioclose, Fioseek

Endtimer	–	Record the ending time of a set of database activities
Eventtime	–	Retrieve the time of an EMPOWER/CS event
Exec	–	Execute the parse
Fetch	–	Fetch the specified data from the database
FetchRaw	–	Fetch the specified binary data from the database
Fioautorewind	–	Automatically rewind the file pointer to the beginning of the specified file
Fioclose	–	Closes a specified file
Fiodelimiter	–	Define field delimiters for a specified file
Fioopen	–	Opens a specified file
Fioreadchar	–	Reads n bytes from a specified file
Fioreadfield	–	Read the next field from a specified file
Fioreadfields	–	Read multiple fields from a specified file
Fioreadline	–	Read the next line from a specified file
Fiorewind	–	Rewind the file pointer to the beginning of the specified file
Fioseek	–	Set the file pointer to a specific byte in the file
Fioshare	–	Identifies a file to be shared
Fioskipchar	–	Skip forward n characters in the file
Fioskipfield	–	Skip forward n fields in the file
Fioskipline	–	Skip forward n lines in the file
Fiounshare	–	Discontinue the sharing of a file
Fiowritechar	–	Write n bytes from the buffer to the file
GetNextRow	–	Sort through the fetched rows of data
GetIntVar	–	Return the current integer value of a specified variable
GetVar	–	Return the current value of a specified variable
gv_add	–	Add the specified amount to the current value of a variable
Gv_add, Gv_addv	–	Add the specified amount to the current value of a variable
Gv_alloc	–	Allocate access to a variable
gv_and	–	Apply bit-wise AND masking to a variable
Gv_and, Gv_andv	–	Apply bit-wise AND masking to a variable
gv_dec	–	Decrease the value of a variable by one
Gv_dec, Gv_decv	–	Decrease the value of a variable by one
gv_div	–	Divide the current value of a variable by the specified amount
Gv_div, Gv_divv	–	Divide the current value of a variable by the specified amount
Gv_free	–	De-allocate access to a variable
gv_getparallel	–	Return the current value of a parallel variable
Gv_getparallel	–	Return the current value of a parallel variable
gv_inc	–	Increase the value of a variable by one
Gv_inc, Gv_incv	–	Increase the value of a variable by one

gv_init	–	Initialize a variable, creating the variable if necessary
gv_lshift	–	Perform a bit-wise shift to the left on a variable
Gv_lshift, Gv_lshiftv	–	Perform a bit-wise shift to the left on a variable
gv_mod	–	Perform a modulo operation on a variable
Gv_mod, Gv_modv	–	Perform a modulo operation on a variable
gv_mul	–	Multiply the current value of a variable by the specified amount
Gv_mul, Gv_mulv	–	Multiply the current value of a variable by the specified amount
gv_or	–	Apply bit-wise OR masking to a variable
Gv_or, Gv_orv	–	Apply bit-wise OR masking to a variable
gv_parallel	–	Wait until the value of the variable becomes greater than zero then decrement the variable by one
Gv_parallel	–	Wait until the value of the variable becomes greater than zero then decrement the variable by one
gv_protect	–	Protect a variable from access by other users and scripts
Gv_protect	–	Prevent other scripts from accessing a variable until the Gv_unprotect() function is executed
gv_read	–	Return the current value of a variable
Gv_read, Gv_readv	–	Return the current value of a variable
gv_rm	–	Remove a variable from shared memory
gv_rshift	–	Perform a bit-wise shift to the right on a variable
Gv_rshift, Gv_rshiftv	–	Perform a bit-wise shift to the right on a variable
gv_seg	–	Control the shared memory segment
gv_setparallel	–	Assign a new value to a parallel variable
Gv_setparallel	–	Assign a new value to a parallel variable
gv_stat	–	Return a list showing the status of one or more variables
Gv_stat	–	Return a structure showing the status of a variable
gv_sub	–	Subtract the specified amount from the current value of a variable
Gv_sub, Gv_subv	–	Subtract the specified amount from the current value of a variable
gv_test	–	Test if the specified comparison is true
Gv_test	–	Test if the specified comparison is true
gv_unparallel	–	Increase the value of the parallel variable by one
Gv_unparallel	–	Increase the value of a parallel variable by one
gv_unprotect	–	Release a variable for access by other users and scripts
Gv_unprotect	–	Remove protection from a variable, allowing other scripts access to the variable
gv_waituntil	–	Wait until the specified comparison is true
Gv_waituntil	–	Wait until the specified comparison is true
gv_waitwhile	–	Wait while the specified comparison is true
Gv_waitwhile	–	Wait while the specified comparison is true
gv_write	–	Assign a new value to a variable
Gv_write, Gv_writev	–	Assign a new value to a variable

gv_xor	-	Apply bit-wise EXCLUSIVE-OR masking to a variable
Gv_xor, Gv_xorv	-	Apply bit-wise EXCLUSIVE-OR masking to a variable
Hostname	-	Specify the name of the host machine
InitialWindow	-	Restore the Windows desktop as captured
Inote	-	Record a Monitor message (integer)
KeyDown	-	Emulate pressing a key
KeyPress	-	Emulate pressing and releasing a key
KeyUp	-	Emulate releasing a key
Language	-	Specify the language to be used
LeftButtonDown	-	Emulate pressing the left button down on the mouse
LeftButtonPress	-	Emulate pressing and releasing the left button on the mouse
LeftButtonUp	-	Emulate releasing the left button
LeftDblPress	-	Emulate two consecutive presses and releases of the left button of the mouse
Log	-	Record a message in the log file
Logoff	-	Close the communication link to the database
Logon	-	Open a communication link to the database
MiddleButtonDown	-	Emulate pressing the middle button down on the mouse
MiddleButtonPress	-	Emulate pressing and releasing the middle button on the mouse
MiddleButtonUp	-	Emulate releasing the middle button on the mouse
MiddleDblPress	-	Emulate two consecutive presses and releases of the middle button of the mouse
Note	-	Record a Monitor message (string)
Open	-	Open a cursor structure
Openenv	-	Open a database environment

Paceconstant	–	Set a constant script pace
Pacetne	–	Set a script pace defined by a truncated negative exponential distribution
Paceuniform	–	Set a script pace defined by a uniform distribution of two values
Parse	–	Parse a SQL statement
Password	–	Specify the user password
Pause	–	Pause a transaction in progress
Pointerrate	–	Set the emulated mouse pointer speed
Range	–	Produce a random number from the specified range
RightButtonDown	–	Emulate pressing the right button down on the mouse
RightButtonPress	–	Emulate pressing and releasing the right button on the mouse
RightButtonUp	–	Emulate releasing the right button on the mouse
RightDblPress	–	Emulate two consecutive presses and releases of the right button of the mouse
Rollback	–	Rollback all database processing
Seed	–	Seed the random number generator
Servername	–	Specify the name of the server
Set	–	Turn on script options
SetIntVar	–	Specify a new integer value for a specified variable
SetVar	–	Specify a new value for a specified variable
Sleep	–	Suspend script execution for an interval
Sql	–	Parse a SQL statement
SqlExec	–	Execute a SQL statement
Suspend	–	Suspend script execution until a resume signal is received
SysKeyDown	–	Emulate pressing a key with the Alt key
SysKeyPress	–	Emulate pressing and releasing a key with the Alt key
SysKeyUp	–	Emulate releasing a key with the Alt key
Think	–	Perform a thinking delay
Thinkactual	–	Define actual think time distribution
Thinkconstant	–	Define constant think time distribution
Thinktne	–	Define a truncated negative exponential think time
Thinkuniform	–	Define uniform think time distribution
Time	–	Get the current UNIX script driver machine time
Timeout	–	Specify timeout condition
Transaction	–	Define a database transaction
Type	–	Emulate keystrokes entered at the keyboard
Typerate	–	Set the emulated typing speed

Unset	–	Turn script options off
Username	–	Specify the name of the user
WindowRcv	–	Paint window on screen

0965794.102600

FUNCTION

AppName

SYNTAX

```
AppName(lognum, appname)
int lognum;
char *appname;
```

DESCRIPTION

Parameters

`lognum`	The number of the log on structure that will access the SUT
`appname`	The name of the client application that will interact with the SUT

Comments

The `AppName()` function is inserted into the script before a log on connection when the name of the client application that will be interacting with the SUT is specified. During script execution, the `AppName()` function is used to define the application name a logon connection will use to interact with the SUT. Application names help the SUT to identify logon connections.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO

Hostname, Language, Password, Servername, Username

FUNCTION **AppWait**

SYNTAX

```
void AppWait(n)
double n;
```

DESCRIPTION

Parameters

n The amount of time in seconds taken to draw a window on screen

Comments

This function is inserted into the script file during Capture to record the amount of time taken to draw a window on screen. It also inlcudes the amount of application processing time for certain database operations. For instance, if the application retrieved and requested the sum of 100 numbers from the database, the total `AppWait()` would include the time taken to add the numbers and the time taken to draw a window.

The `AppWait()` function is used during script execution in Non-Display mode to emulate application delay for drawing windows and application processing before the emulated user can continue to the next input. This function does not apply to Display mode script execution because the application actually processes database operations.

If you wish to change your AppWait delay, you may set a multiplication factor with the `AppWaitFactor()` function.

This function multiplies the AppWait delay times the factor. For instance, if during script execution, you think your application may be twice as slow because it is receiving twice as much data from the server, you can set the `AppWaitFactor()` to 2 so that the script will emulate the extra delay.

RETURN VALUE (not applicable)

EXAMPLES

In the following example, the script recorded an application delay of 1.16 seconds to draw the `Program Manager` window.

```
AppWait(1.16);
WindowRcv("SfDwAcSfSfSfPt");

CurrentWindow("Program Manager");
```

SEE ALSO

AppWaitFactor, WindowRcv

FUNCTION	AppWaitFactor
SYNTAX	`AppWaitFactor(n)` `double n;`

DESCRIPTION

Parameters

`n` The multiplying factor for the value of the AppWait delay. The default value of `n` is 1.

Comments

The `AppWaitFactor()` function specifies a multiplying factor for the value of the AppWait delay.

You may insert the `AppWaitFactor()` function into a script when editing. This function applies only to `AppWait()` functions that occur after the `AppWaitFactor()` function in the script. Multiple `AppWaitFactor()` functions may be inserted into a single script.

The `AppWaitFactor()` function is particularly useful when an increased load on the server could cause increased AppWait delays where none occurred in previous script executions.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example doubles the `AppWait()` for closing the `General Ledger` window, and sets the `AppWait()` back to 1 for opening the `Accounting` application:

```
CurrentWindow("General Ledger",20,30,234,234);

ButtonPush("Close",254,261);
AppWaitFactor(2);

AppWait(1.6);
WindowRcv("SfDwAcSfSfSfPt");

CurrentWindow("ProgramManager",0,0,1048,1048);

ButtonPush("Accounting Application",23,45);

AppWaitFactor(1);

AppWait(2.1);
WindowRcv("SfCwCwAcSfSfSfSfPt");

CurrentWindow("Accounting",20,40,234,234);
```

SEE ALSO AppWait

FUNCTION **Beginfunction**

SYNTAX

```
Beginfunction(str)
char *str;
```

DESCRIPTION

Parameters

`str` The name of the function as defined by the EMPOWER/CS user. This parameter is null-terminated character string.

Comments

The `Beginfunction()` function defines the start of a set of emulated activities called a function.

EMPOWER/CS allows you to place C language functions in the script during Capture by selecting the **Function** button in the Capture window. When you define a function, EMPOWER/CS automatically inserts `Beginfunction()` and `Endfunction()` around the function in the script file. You also may insert these functions when editing your script file.

The functions `Beginsource()` and `Endsource()` automatically are placed around `Beginfunction()` and `Endfunction()` to prepare the specified function as a source file. For instance, during a multi-user emulation, you may wish to break a common function out into one source file that can be called by multiple scripts. `Beginsource()` and `Endsource()` specify a file as a source file.

When a script is executed, EMPOWER/CS places a time stamp with each `Beginfunction()` and `Endfunction()` function so that the Report tool can calculate response time statistics for the function. The response time data for these functions represents the time required to perform all interactions within the function.

Note that the "function" identified by the `Beginfunction()` and `Endfunction()` functions is different from the C language concept of a

function. C language functions, which are similar to procedures in Pascal and subroutines in FORTRAN, are used to group a set of EMPOWER/CS functions and C statements together. They are called during script execution.

RETURN VALUE (not applicable)

EXAMPLES In the following example, the `Beginfunction()` and `Endfunction()` statements are placed in a C language function that consists of opening a window. The name of the function is `openwindow`. A function call is placed within the scenario and the actual function is listed after `Endscenario()`:

```
Empower();
{
AppWait(0.33);
WindowRcv("SfSfSfPt");

...

openwindow();

LeftButtonDown(132,91);
LeftButtonUp(158,212);

AppWait(3.52);
WindowRcv("ScSfSfSfPt");

Endscenario("example");
}

openwindow()
{
Beginsource("script1);
Beginfunction("openwindow");
```

(continued on following page . . .)

```
LeftDblPress(438,304);
LeftDblClick(438,304);


AppWait(0.24);
WindowRcv("PtCoCwCwDwAcSfCwCoSfCwCwCwCwSfAcPtPtPt");
WindowRcv("Pt");

CurrentWindow("New...");

/* Clicked (Button) (Cancel) */
LeftButtonDown(342,256);


Endfunction("openwindow");
Endsource();
}
```

Timestamps in the log file will correspond to the `Beginfunction()` and `Endfunction()` statements in the script.

Example:

```
>>>     12 Beginfunction("openwindow") 14:43:23.01
. . .
>>>     27 Endfunction("openwindow") 14:43:27.15
```

SEE ALSO Beginsource, Endfunction, Endsource

FUNCTION **Beginscenario**

SYNTAX

```
Beginscenario(str)
char *str;
```

DESCRIPTION

Parameters

`str` The name of the script as defined in the Capture session. This name is a null-terminated character string.

Comments

The term "scenario" generally applies to a large portion of emulated activity. Usually for EMPOWER/CS, a scenario is an entire script. EMPOWER/CS automatically places `Beginscenario()` and `Endscenario()` functions at the beginning and end of script activity during Capture.

If you wish to change the `str` parameter for this function, you may do so when editing your script.

EMPOWER/CS provides summary performance information for each scenario. For example, the Report tool provides scenario start and stop times, duration, throughput, and average response times.

By default, the duration of the test is determined by the time stamps of the first `Beginscenario()` and of the last `Endscenario()`.

RETURN VALUE (not applicable)

EXAMPLES The name of the scenario is taken from the name of the script source file. Initiation of capturing the script `example` would cause the following functions to be inserted at the beginning and end of the script:

```
Beginscenario("example")
...
Endscenario("example")
```

The log file created from script execution will contain time stamps for the beginning and end of the scenario.

Example:

```
>>> 10 Beginscenario("example") 12:05:29.00
...
>>> 462 Endscenario("example") 12:08:23.07
```

SEE ALSO Endscenario

08697954.102600

FUNCTION **Beginsource**

SYNTAX

```
Beginsource(str)
char *str;
```

DESCRIPTION

Parameters

`str` The name of the script source file which is a null-terminated character string

Comments

Modular script design is achieved by storing one or more functions in separate script source files, which allows a script to include a function call rather than repeating a set of interactions. When each source file is compiled with the `-c` option of the `cscc` command, an object file for each script is created which can be compiled with and linked to the main script file.

If a script is compiled from multiple source files, each source file should include `Beginsource()` and `Endsource()` statements. `Beginsource()` and `Endsource()` specify the source file used for script execution.

When editing your scripts, you should insert the `Beginsource()` function at the source file's entry point, typically just before the first executable statement in each function. The `Endsource()` function should be placed at file's exit point, typically just after the last executable statement in each function.

`Beginsource()` and `Endsource()` also are placed around the C functions defined during Capture. Functions are formatted in this way so that for a multi-user emulation you may break the function out of a script into a separate source file to be used by multiple scripts.

The `Beginsource()` and `Endsource()` statements are used in Monitor to indicate the source file being executed at a certain point of script execution.

RETURN VALUE (not applicable)

EXAMPLES In the following example, elements of a function called `logoff1()` may be contained in a separate script file called `exitapp.c`, as shown:

```
logoff1()
{
Beginsource("exitapp");
Beginfunction("logoff1");

Think(9.05);

LeftButtonDown(207,97);
LeftButtonUp(226,241);


AppWait(0.33);
WindowRcv("ScDwAc");

CurrentWindow("Capture - script1",21,690,57,726);

Commit(LOG1);


Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endfunction("logoff1");
Endsource();
}
```

SEE ALSO Endsource

FUNCTION **Begintimer**

SYNTAX

```
Begintimer(str)
char *str;
```

DESCRIPTION

Parameters

`str` A string that specifies the name of the timer to begin

Comments

`Begintimer()` and `Endtimer()` are used in a script to measure script activity. These functions are time stamped in the executed script's log file and are used by Report for measuring response time of the specified activity. `Begintimer()` must have a corresponding `Endtimer()` function and the corresponding functions must have the same `str` parameter.

The `Begintimer()` and `Endtimer()` functions can be nested.

If the Capture option **Insert timer** is selected, the `Begintimer()` and `Endtimer()` functions are inserted automatically into the script by EMPOWER/CS to measure response time of database traffic. These functions will be inserted around database traffic that occurs between two user events. A user event such as pressing the Return key on the keyboard or activating a pushbutton on screen may initiate database activity. The `str` parameter identifies the user event that initiated the database activity. For example, if a pushbutton initiated database activity, the parameter to `Begintimer()` and `Endtimer()` would list a two-level tree structure that lists the parent window and the button.

The `Endtimer()` function will occur at the end of the database activity when a window is drawn on screen or another set of user input begins.

These functions also can be user-specified. Similar to the functions `Beginfunction()` and `Endfunction()`, they can be inserted manually into the script when editing or by activating the EMPOWER/CS **Timer**

button in the **Capture** window during Capture. If you select the **Timer** button, `Begintimer()` is inserted into the script measuring activity until the **End** button is selected to insert `Endtimer()`.

When editing your script, you may change the `str` parameter to a string that is more meaningful for your emulation.

RETURN VALUE (not applicable)

EXAMPLES In the following example, `Begintimer()` and `Endtimer()` were inserted into the script around database activity. Notice that the parameter to these functions lists the last user event, which was pressing the **OK** button in the window titled **Status**.

```
CurrentWindow("Status",240,180,408,301);
ButtonPush("OK",322,267);

WindowRcv("SfAcSfDw");

...

Begintimer("Status_OK");

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endtimer("Status_OK");
```

SEE ALSO Endtimer

FUNCTION BeginTransaction

SYNTAX

```
BeginTransaction(lognum)
int lognum;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

Comments

The `BeginTransaction()` function is used to start a transaction in a script. This function is inserted into the script when the client application instructs the database to begin a transaction.

The transaction begun with `BeginTransaction()` can be ended with a `Commit()` or `Rollback()` function. It also can be paused and continued with the functions `Pause()` and `Continue()`.

RETURN VALUE If the function succeeds, a zero is returned. If the function fails, a -1 is returned.

EXAMPLES The following example defines a transaction:

```
BeginTransaction(LOG1);
Open(LOG1,CUR1);
...
Exec(CUR1);
Commit(LOG1);
```

SEE ALSO Commit, Continue, Pause, Rollback

FUNCTION **Bind**

SYNTAX

```
Bind(curnum, name, type, length)
int curnum;
char *name;
int type, length;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure of the associated SQL statement
`name`	The variable's placeholder name as listed in the SQL statement
`type`	The variable's data type
`length`	The length of the variable in bytes

Comments

If a SQL statement requires data to be input to the database, placeholders for input variables will be listed in the SQL statement and are indicated by leading colons. A `Bind()` function will be inserted into the script for each placeholder that is listed. During script execution, the `Bind()` function binds the placeholder to a variable that is to be passed to the database. For example, if a select-list item of the SQL statement includes a placeholder such as `:NAME`, the `Bind()` function is inserted into the script to bind `:NAME` to a variable.

`Bind()` associates the address of a script variable with the specified select-list item, or placeholder, in the SQL statement. The parameters of the `Bind()` function bind the variable to its placeholder by a specific cursor number, the placeholder name listed in the SQL statement, the variable's data type, and the variable's length.

The `Bind()` function is called after the `Parse()` statement and before `Exec()`.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client application and SUT and therefore, break the script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

If an input variable is specified within a `Parse()` statement, the `Bind()` function is inserted into the script instead of `Define()`. The following example demonstrates a `Bind()` function inserted for the variable, `X`, which was listed in the preceding SQL statement:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP WHERE EMPNO=:X");

. . .

Bind(CUR1, "X", STRING, 10);
```

SEE ALSO

Bindp, BindDefine, Define

FUNCTION **BindDefine**

SYNTAX

```
BindDefine(curnum, name, type, length)
int curnum;
char *name;
int type, length;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure of the associated SQL statement
`name`	The variable's placeholder name as listed in the SQL statement
`type`	The variable's data type
`length`	The length of the variable

Comments

A `BindDefine()` function is inserted into the script when data is passed to and returned from the database as designated in a SQL statement. During script execution, `BindDefine()` inputs a specified variable to the database and then returns the value of the variable.

`BindDefine()` associates the address of a script variable with the specified select-list item, or placeholder, in the SQL statement. This placeholder is designated by leading colons. The variable is defined by the parameters of the `BindDefine()` function which identify it by a specific cursor number, the placeholder name in the SQL statement, the variable's data type, and the variable's length.

The `BindDefine()` function must be called after the `Parse()` statement and before `Exec()`.

As an example, suppose the SQL statement specified that the value for `:NAME` was Smith and it was to be inserted into a record overwriting the

existing value, Jones. `BindDefine()` would specify Smith as the new value for the record and then return the old value, Jones, to the variable.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client application and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Bind, Bindp, Define

FUNCTION **Bindp**

SYNTAX

```
Bindp(curnum, pos, type, length)
int curnum;
char *pos;
int type, length;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure of the associated SQL statement
`pos`	A position index for the variable's placeholder as listed in the SQL statement
`type`	The variable's data type
`length`	The length of the variable

Comments

This function may be inserted into the script instead of a `Bind()` function when a variable referenced in a SQL statement contains data to be input to the database. Instead of binding a placeholder name to the variable, the `Bindp()` function binds a variable's position index in a SQL statement to the variable.

`Bindp()` associates the address of a script variable with the specified select-list item, or placeholder, in the SQL statement. This placeholder is indicated by leading colons. The parameters of the `Bindp()` function identify the variable by a specific cursor number, the position index in the SQL statement of the variable's placeholder, the variable's data type, and the variable's length. The positions of the select list-items start at "1" for the first (or left-most) select-list item, "2" for the second, etc.

The `Bindp()` function must be called after the `Parse()` statement and before the `Exec()` function.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client application and SUT and therefore, break the script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates Bindp() in a script file:

```
Parse(CUR1, "select ename from employee_table where
empno=:empno and deptno=:deptno");

...

Bindp(CUR1, 1, STRING, 20);     /* pos 1 is :empno */
Bindp(CUR1, 2, LONG, 4);  /* pos 2 is :deptno */
```

SEE ALSO

Bind, BindDefine, Define

FUNCTION **ButtonPush**

SYNTAX

```
int ButtonPush(str,x,y)
char *str;
unsigned int x,y;
```

DESCRIPTION

Parameters

`str` The name of the pushbutton or the tree structure that lists the pushbutton

`x` The on screen x coordinate of the pushbutton

`y` The on screen y coordinate of the pushbutton

Comments

The `ButtonPush()` function is inserted in the script file during Capture to indicate that a MS Windows pushbutton was activated (such as **OK**, **Cancel**, **Yes**, **No**, etc.). This function is used during script execution in Display mode to move the mouse to activate a pushbutton defined in the parameter `str`. In Non-Display mode, this function simulates mouse movement as defined in the `x,y` parameters to allow for mouse pointer delay.

The format of the `str` parameter is designed so that EMPOWER/CS can easily locate the specified pushbutton during script execution in Display mode. This format is based on the MS Windows concept of a tree structure and may appear similar to the following:

```
"Tools|#c1|#c4"
```

The MS Windows tree structure is based on a heirarchy of windows where each window that is accessed from a primary, or parent, window is a child of that parent. The `str` parameter is listed right to left from child to parent window where the right-most item is the name of the button. If one of the windows or the button has no name, something like

"#c1" will be listed to indicate the window is a certain numbered child of the preceding parent window.

In the example listed above, #c4 was the button activated and is the fourth child of the first child (#c1) of the Tools window.

If a button was activated in the active, or current window, only the button will be listed in the str parameter.

If you wish to edit this function in your script file, you can use the Tree Tool under EMPOWER/CS Tools to determine the tree structure for a particular pushbutton.

RETURN VALUE

If successful, the ButtonPush() function returns 1. If unsuccessful, the function will return a zero.

EXAMPLES

In the following example script segment, the user pushed the button OK in the current window, "Run", to open an application:

```
CurrentWindow("Run");
Type("c:\\acct\\c\\acct^M);
ButtonPush("OK",354,153);


AppWait(0.05);
WindowRcv("CwCwCwCwAcSfCwPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
```

FUNCTION **Cancel**

SYNTAX

```
Cancel(num, opt)
int num, opt;
```

DESCRIPTION

Parameters

`num`	An identifier of the logon or cursor structure for which operations are to be cancelled
`opt`	An option of either `ALL` or `CURRENT`

Comments

This function may appear throughout your script and is inserted into the script when database operations were cancelled for either a logon or cursor structure. For example, at some point during the Capture session, the user may have requested that the current operation of fetching a record was cancelled.

During script execution, the `Cancel()` function cancels operations in progress for the specified structure without closing the structure. The `opt` parameter specifies an option of cancelling all operations or the current operation on the specified structure.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates the `CURRENT` option of `Cancel()`. If, during Capture, all desired data has been fetched for a query, the user may wish to cancel the query before it has completed.

Instead, the user may cancel the operation (which saves database processing time during script execution). Such an operation would be captured into the script as:

```
Cancel(CUR1, CURRENT);
```

The following example demonstrates using the option ALL. Suppose a logon connection, LOG1, contained ten cursors. The following function would cancel all ten cursors:

```
Cancel(LOG1, ALL);
```

FUNCTION **Close**

SYNTAX

```
Close(curnum)
int curnum;
```

DESCRIPTION

Parameters

`curnum` An identifier of the cursor communication structure to be closed

Comments

A `Close()` function is inserted into the script when a cursor is closed. During script execution, `Close()` closes the cursor communication structure that was opened with the associated `Open()` function. Once a cursor is closed, no additional processing (i.e., `Parse()`, `Bind()`, `Define()`, `Describe()`, `Exec()`, etc. functions) can be performed on that cursor.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES In the following script example, `Close()` will close the cursor, `CUR1`, before the executed script exits:

```
Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE1);

Endscenario("script1");
```

SEE ALSO Open

00526074.102601

FUNCTION Closenv

SYNTAX

```
Closenv(dbnum)
int dbnum;
```

DESCRIPTION

Parameters

`dbnum` A database environment number

Comments

`Closenv()` closes the environment opened with the associated `Openenv()`. This function is inserted into the script when a database environment is closed.

After a database environment is closed, no logon connections can be made within the specified environment.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example script segment demonstrates that `Closenv()` will close the database environment `ORACLE1` before the executed script exits:

```
Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE1);

Endscenario("script1");
```

SEE ALSO Openenv

FUNCTION CmpVar

SYNTAX

```
CmpVar(curnum, var, value)
int curnum;
char *var, value;
```

DESCRIPTION

Parameters

`curnum`	An identifier of a cursor communication structure
`var`	The variable used for the comparison
`type`	The data type of the variable
`length`	The length of the variable in bytes
`value`	The specified value for the comparison

You may insert the `CmpVar()` function into your script to compare a variable from a SQL statement to a specified value.

This function determines if the specified variable, `var`, in the current row is equal to the value specified in the `value` parameter. The address of the variable must be passed to `CmpVar()`.

This function could be used for fetching records until a desired value is returned.

RETURN VALUE If the variable is equal to the specified value a zero is returned; if the variable is greater than the specified value, 1 is returned; and, if the variable is less than the specified variable a -1 is returned.

EXAMPLES

An example of this function follows:

```
Parse(CUR1, "select ename from employee_table");

Define(CUR1, "1", STRING, 50);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);
while (CmpVar(CUR1, "1", "Smith") != 0){
  Fetch(CUR1);
  GetNextRow(CUR1);
}
```

SEE ALSO

GetIntVar, GetVar, SetIntVar, SetVar

FUNCTION **Commit**

SYNTAX

```
Commit(num)
int num;
```

DESCRIPTION

Parameters

`num` An identifier of a logon or cursor communication structure

Comments

The `Commit()` function may appear throughout the script and is inserted into the script when database processing is committed to the database. During script execution, this function commits to the database all processing the script has completed(i.e., updating records, deleting records, adding records, etc.) since processing was last commited. The parameter `num` specifies a logon or cursor number for which operations are to be committed.

The `Rollback()` function is inserted into the script if script operations are not commited to the database but are rolled back. `Rollback()` restores the database to its original state before a script was executed.

Note: Because these functions are inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit the functions or remove them from the script. If you change these function from when they were captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates that all database processing for `LOG1` will be committed to the database before the executed script exits:

```
Commit(LOG1);

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE1);


Endscenario("script1");
```

SEE ALSO Rollback

FUNCTION **Continue**

SYNTAX

```
Continue(lognum, transnum)
int lognum, transnum;
```

DESCRIPTION

Parameters

`lognum`	An identifier of a logon communication structure
`transnum`	An identifier of the transaction to be resumed

Comments

The `Continue()` function specifies an application-defined database transaction that is to be continued. This function is captured into the script when the client application instructs the database to resume execution of the paused transaction. It will be inserted after the associated `BeginTransaction()` and `Pause()` functions.

The specified `transnum` parameter must correspond to a `transnum` specified in an associated `Pause()` function. A transaction can be continued only if no other transactions are currently running on the specified logon connection.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates this function in a script file:

```
BeginTransaction(LOG1);

....
Pause(LOG1, TRANS1);

BeginTransaction(LOG1);

....
```

(continued on following page . . .)

```
Pause(LOG1, TRANS2);

Continue(LOG1, TRANS1);
Commit(TRANS1);

Continue(TRANS2);
Commit(TRANS2);
```

SEE ALSO BeginTransaction, Pause

FUNCTION CurrentWindow

SYNTAX

```
CurrentWindow(window, left, top, width, height)
char *window;
int left, top, width, height;
```

DESCRIPTION

Parameters

`window`	The title of the active window
`left,top`	The left, top x,y coordinates of the window on screen
`width,height`	The size x,y coordinates of the window on screen

Comments

The `CurrentWindow()` function is captured into your script file to record all titled windows that were activated during Capture. This function is used to ensure windows are in their captured state during script execution in Display mode. It designates a window currently active and moves the specified window to its captured position. The `CurrentWindow()` function also is used during Monitor sessions to add context to script execution in Display and Non-Display modes.

The parameter `window` identifies the title of the current window. The `top,left` and the `width, height` coordinates indicate the on screen coordinates of the specified window. If the `width, height` coordinates are `0,0`, the window is in a minimized state. If the `top,left` x,y coordinates are `0,0` and the `width, height` coordinates are `MAXWIDTH,MAXHEIGHT`, the window is maximized. If none of these conditions apply, the window is considered to be in a normal state.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the

behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE (not applicable)

EXAMPLES In the following example, `CurrentWindow()` designates that the `Program Manager` window was captured in a normal state:

```
AppWait(4.34);
WindowRcv("SfAcPtSfPtPtSfDwAcSfSfSfPt");

CurrentWindow("Program Manager",413,679,1029,771);

KeyPress(VK_CONTROL);
KeyPress(VK_F12);

LeftButtonPress(443,112);
```

SEE ALSO InitialWindow

FUNCTION Data

SYNTAX

```
Data(curnum, str, .../* args */)
int curnum;
char *str, *args;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure of the associated SQL statement
`str`	The data to be inserted into the database
`arg`	Optional variable arguments

Comments

The `Data()` function is inserted into the script when data is specified to be input to the database. The `str` parameter is pipe delimited and lists the data for each input variable that was defined with `Bind()`. Therefore, the `Data()` function will follow a `Bind()`, `BindDefine()`, or `Bindp()` function and also will occur before an `Exec()`.

During script execution, `Data()` specifies the data to be inserted into or selected from the database.

The `Dbset()` option `INSERTSIZE` applies to the `Data()` function in that the number of `Data()` functions inserted into the script will correspond to the value of `INSERTSIZE`. `INSERTSIZE` specifies the number of rows of data to be inserted into the database.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `Data()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments

must be string pointers. The string conversion specification is provided by the characters %s.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demontrates a SQL statement and its corresponding data line in a script:

```
/* insert data into database */
Parse(CUR1, "insert empno, ename, empjob into emp_table");

Bind(CUR1, "empno", INT, 4);
Bind(CUR1, "ename", STRING, 30);
Bind(CUR1, "empjob", STRING, 20);

/* 123 refers to empno, Smith -- ename, typist -- empjob */
Data(CUR1, "123|Smith|typist");

Exec(CUR1);
```

The parameter to `Data()`, "`123|Smith|typist`" represents the data that was inserted into the database. `123` would correspond to `EMPNO`, `Smith` would correspond to `ENAME`, and `typist` would refer to `EMPJOB`.

The following example demonstrates using this function to accept a variable argument:

```
Parse(CUR1, "select * from emp_table where lname = :name and empjob = :job");

Bind(CUR1, ":name", STRING, 30);
Bind(CUR1, ":job", STRING, 30);

/* this Data() will get name from file and alway use job of typist */
Data(CUR1, "%s|typist", Fioreadfield(datafile));

Exec(CUR1);
```

FUNCTION **Dberror**

SYNTAX
```
Dberror(cond)
int cond;
```

DESCRIPTION *Parameters*

`cond` The condition, either `CONTINUE` or `EXIT`, for database errors

Comments

Similar to the `Timeout()` function, `Dberror()` specifies the condition or action to be taken when a script encounters a database error. Valid conditions are `CONTINUE` and `EXIT` where the script will either continue script execution or exit the script when a database error is encountered.

The default function `Dberror(CONTINUE)` is placed at the top of every script created by EMPOWER/CS. During script execution, `Dberror(CONTINUE)` specifies that the script will continue if it enounters a database error. You may edit this function and/or insert multiple `Dberror()` functions in the script file to suit your testing needs.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates the `Dberror()` function set to `CONTINUE`:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */


Typerate(5);               /* Typing delay in CPS */
Thinkuniform(1,2.5);       /* Think delay */
Seed(getpid());     /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE);         /* What to do on Database errors */
Unset(NOTIFY);      /* Don't display warnings. I'll use Mon to find them */
```

SEE ALSO   Timeout

FUNCTION **Dbset**

SYNTAX

```
Dbset(num, opt, value)
int num, opt, value;
```

DESCRIPTION

Parameters

`num`	Specifies an environment, logon structure, or cursor for which options are being set
`opt`	Specifies the option being set
`value`	Sets a value for the option, either a numerical or string value, or `TRUE` or `FALSE`

Comments

The `Dbset()` function is used in a script to set certain features for a particular database environment, logon structure, or cursor. These features will apply to all subsequent operations in the specified structure. `Dbset()` is inserted into your script according to the behavior of your client application and the SUT. The options set in this function are dependent on the client application. Because these options are not user-defined, but occur specific to the database and client application, the `Dbset()` function generally should not be edited or removed from your script file.

The `Dbset()` options that commonly will appear in your scripts are listed below:

`FETCHSIZE`

Specifies the number of records to fetch from the database when a `Fetch()` function is executed. This number will be less than or equal to the value specified in the `MAXARRSIZE`. The `Dbset()` function that sets the `FETCHSIZE` will occur before a `Fetch()`. The `FETCHSIZE` value can not be larger than the `MAXARRSIZE`. If the `FETCHSIZE` was specified as 50 and `MAXARRSIZE` was specified as 20, EMPOWER/CS will reduce the `FETCHSIZE` to 20.

MAXARRSIZE

Sets the maximum array size for retrieving or inserting records. If the array size is set at 20, 20 rows of data can be inserted or fetched. This option set at 20 allocates 20 placeholders for inserting or fetching 20 rows of data. The `Dbset()` function that sets the `MAXARRSIZE` will occur before a the `Bind()` and `Define()` functions in a script.

INSERTSIZE

Specifies the number of rows of data to be inserted into the database. This option will be listed before the `Data()` function in a script in the format `Dbset(CUR1, INSERTSIZE, n)`. This option allows array binding. For example, if `n` is 0 or 1, one row can be inserted at a time into the database. If `n` is 50, 50 rows will be inserted into the database and 50 data lines will be listed for every row before `Exec()`. The `INSERTSIZE` value can not be larger than the specified `MAXARRSIZE`.

OFFSET

Specifies the offset for inserting records in an array. This option is used with the `INSERTSIZE` option. If an array includes four names and the `OFFSET` is set to 2, then inserting rows will start from the second position with the second name. The `Dbset()` function specifying this option will occur before an `Exec()` function. If the `OFFSET` is specified as larger than the `MAXARRSIZE`, a database error will occur.

WAITRES

Specifies whether or not to wait for resources. If this option is set to `TRUE`, the script will wait indefinitely for requested information from the database. If it is set to `FALSE` and the script does not receive the requested information, the script will receive an error that the resource was not available. Script execution will then either continue or exit based on the condition set in `Dberror()`.

DEFER

Specifies whether or not to defer the `Parse()` statement. If this option is set to `TRUE`, a deferred parse will be performed when the script encounters the `Parse()` function. If the option is set to `FALSE`, a normal `Parse()` is executed.

Normally, the SQL statement is sent to the database when the script encounters `Parse()` and is processed to ensure it is semantically correct. The SQL statement is stored waiting for the `Exec()` call that actually will execute it. If `DEFER` is set to true, the SQL statement is not sent to the database until the script encounters an operation that requires input from the database such as `Exec()` or `Describe()`.

Other `Dbset()` options may appear in your scripts. A full list of all possible options follows. This list is divided into those options that set integer values and those that require TRUE or FALSE values:

Integer Value

Option	Description
FETCHSIZE	number of rows to be fetched
MAXARRSIZE	max number of rows to be fetched
INSERTSIZE	number of rows to be inserted
OFFSET	row number where to start inserting
MAXCONNECT	maximum number of connections in an environment
PACKETSIZE	maximum packetsize on log
ROWCOUNT	maximum number of rows to return
TEXTSIZE	limits size of text or image data
ISOLATIONLEVEL	transaction isolation level
AUTHOFF	turns specified authorization off
AUTHON	turns specified authorization on
CURREAD	security label spec. cur read level
CURWRITE	security label spec. cur write lev
DATEFIRST	which day of week is first
DATEFORMAT	format of date
IDENTITYOFF	disable inserts into table ident column
IDENTITYON	enable inserts into table ident column

TRUE or FALSE

Option	Description
DEFER	deferred parse
UPDATE	cursor is for updating
READONLY	cursor is read only
SAVEOPTS	do not deallocate the structure
FORCELOGOFF	force logoff

FORCEEXIT	force exit
RECOMPILE	recompile stored procedure before executing
RETPARAM	return parameter
TRUNCATE	truncate data
INTERRUPT	application can be interrrupted
ANSI_BINDS	force ansi style binds
DEFER_IO	deferred io
ASYNC_IO	async io
SYNC_IO	sync io
EXTRA_INF	return extra information for error
EXPOSE_FMTS	expose formats
BULKCOPY	allow bulk copy on connection
ASYNCNOTIFICATION	asynchronous notification
DIAG_TIMEOUT	what to do on a timeout
APPSECURITY	application-defined security
SYBSECURITY	sybase-defined security
ENCSECURITY	encrtyped security
TRUSTSECURITY	trusted security
ANSINULL	ansi style nulls
ANSIPERM	ansi style permissions
ARITHABORT	what to do on arithmetic errors
ARITHIGNORE	what to do on arithimetic div. by 0
CURCLOSETRAN	close all cursors on end transaction
NONSTANDSQL	flag non standard sql
FORCEPLAN	force plan or not
FORMATONLY	only send format for data
GETDATA	returns information on every sql statement
NOROWCOUNT	row count
PARSEONLY	only check syntax, do not execute
QUOTEDIDENT	all double quotes signify identifiers
PARSETREES	returns parse resolution trees
SHOWPLAN	generates a description of procedure plan
IOSTATS	return internal io statistics
TIMESTATS	return time statistics
IGNORETRUNC	ignore truncation errors
AUTOCOMMIT	commit on every Exec()
WAITRES	wait for resources instead of error
SPECIAL	sybase version of cursor
HIDDEN_KEYS	expose hidden keys

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

In the following example, the client application specified that the `DEFER` option be set to `TRUE` so that a deferred `Parse()` will occur:

```
Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");
```

In this next example, the `MAXARRSIZE` is set to a size 64 array for the subsequent script operations:

```
Dbset(CUR1,MAXARRSIZE,  64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);
```

FUNCTION **Define**

SYNTAX

```
Define(curnum, pos, type, length)
int curnum;
char *pos;
int type, length;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor of the associated SQL statement
`pos`	The position of the select-list item in the SQL statement
`type`	The variable's data type
`length`	The length in bytes of the variable being defined

Comments

The `Define()` function is inserted into the script during a query when output variables are defined for storing data fetched from the database. During script execution, `Define()` defines output variables for each select-list item listed in a SQL query statement. The SUT places the requested data in these output variables when a `Fetch()` function is later called.

The `Define()` function associates an output variable with each select-list item in a SQL query statement. Each select-list item is designated in the parameters to `Define()` by a cursor number, the item's position in the SQL statement, the variable's data type, and the variable's length. The positions defined in the `pos` parameter begin with 1 for the first (or left-most) select-list item, 2 for the second, etc.

The `Define()` function is called after a `Parse()` statement and before `Fetch()`.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to

edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

In the following script segment, the `Define()` functions define variables for each select-list item of the `Parse()` statement. Notice in the first `Define()` statement below, the variable "1" refers to `EMPNO` in the SQL statement:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO
FROM EMP, UPDATE OF EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,
DEPTNO");
Define(CUR1, "1", CHAR, 5);        /* pos1 = EMPNO */
Define(CUR1, "2", STRING, 11);     /* pos2 = ENAME */
Define(CUR1, "3", STRING, 10);     /* pos3 = JOB */
Define(CUR1, "4", STRING, 5);      /* pos4 = MGR */
Define(CUR1, "5", STRING, 10);     /* pos5 = HIREDATE */
Define(CUR1, "6", LONG, 4);        /* pos6 = SAL */
Define(CUR1, "7", STRING, 9);      /* pos7 = COMM */
Define(CUR1, "8", INT, 4);         /* pos8 = DEPTNO */
Exec(CUR1);
```

SEE ALSO

Bind, Bindp, BindDefine

FUNCTION **Describe**

SYNTAX

```
Describe(curnum, pos)
int curnum, pos;
```

DESCRIPTION

Parameters

`curnum` An identifier of the cursor structure of the associated SQL statement

`pos` The position of the variable in the SQL statement

Comments

During Capture, if a client application requests from the SUT a description of a variable in the SQL statement, a `Describe()` function is inserted into the script. The `Describe()` function is used only for database queries.

During script execution, the `Describe()` function sends a request to the SUT for a description of a specified variable. This variable is specified by a cursor number and by the variable's position in the associated SQL statement. Information returned about a variable may include the name of the variable, the variable's data type, the length of the variable, whether the data in the variable is null-terminated or updateable, etc. This information is used for converting, displaying, or storing the data that will be returned for a query.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates how `Describe()` would appear in a script:

```
Parse(CUR1, " select ename from emp_table");
Describe(CUR1, 1); /* describes ename */
```

The following is an example of the output for `Describe()` which is recorded in the script's log file:

```
>>>     Describe(CUR1, 1);
        size=20, type=CHAR, name=ename, namelen=10, dsize=0, prec=0,
        scale=0, nullok=0
```

SEE ALSO

DescribeAll, DescribeProc

FUNCTION	**DescribeAll**
SYNTAX	`DescribeAll(curnum, pos1, pos2)` `int curnum, pos1, pos2;`

DESCRIPTION

Parameters

`curnum`	An identifier of a cursor structure of the associated SQL statement
`pos1`	The starting position in the SQL statement for a Describe operation
`pos2`	The ending position in the SQL statement for a Describe operation

Comments

A `DescribeAll()` function is inserted into the script when a `Describe()` operation was performed for each select-list item specified between two positions of the SQL statement. This function describes each select list item, or variable, starting from the position specified in `pos1` to the position specified in `pos2`.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

In the following example, the variables listed in the SQL statement from `CUR1`, starting from the first position to the twelfth position, which will be described:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);
```

SEE ALSO Describe, DescribeProc

FUNCTION **DescribeProc**

SYNTAX

```
DescribeProc(lognum, procname)
int lognum;
char *procname;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`procname` The name of the procedure being described

Comments

This function is inserted into the script during Capture when the client application sends a request to the database for a description of a procedure. `DescribeProc()` performs a `Describe()` operation on all the variables referred to in a specified procedure returning information about the variables used in that procedure.

A logon structure is specified in the parameter instead of cursor because the stored procedure must be described before a parse is completed. Therefore, the `DescribeProc()` function will be listed in the script before the `Open()` and `Parse()` functions. A stored procedure is an operation that is stored on the database to be executed later.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the behavior of the client and SUT and, therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Describe, DescribeAll

FUNCTION **Difftime**

SYNTAX

```
double Difftime(first, second, diff);
struct timevalue *first, *second, *diff;
```

DESCRIPTION

Parameters

`first` The first time stamp value

`second` The second time stamp value

`diff` Variable that stores the difference between `first` and `second`

Comments

You may insert this function into your script when editing. It is used to ensure that certain activities in a script occur in a specified amount of time.

`Difftime()` computes the difference in seconds between the `first` and `second` values and stores the result in the variable `diff`. The time difference is returned in a double value. This allows a time difference to be expressed as a floating point (fractional) number, as in 1.5 (one and a half seconds). Normally, the first value is earlier than the second.

The `first`, `second`, and `diff` values are time stamps of `struct timevalue` type. The variable `struct timevalue` is defined in `$EMPOWER/h/empower.h` as:

```
struct timevalue {
      long sec;   /* seconds since Jan 1. 1970 */
      short hsec; /* and hundredths of a second */
}
```

RETURN VALUE The return value is positive if the `first` time value is earlier than the `second` time value. Otherwise, the return value is negative.

EXAMPLES

This example script segment measures the time it takes to enter information into a field and press the left mouse button. One of the past uses of the `Difftime()` function was to help pace a script to provide a required transaction throughput. This operation now is accomplished with Empower's Pace functions.

```
char buf[10];
double difftm;
struct timevalue time1, time2;

...

Time(&time1);
Type("1234^M");

LeftButtonPress(282,174);


AppWait(0.05);
WindowRcv("SfSfSfSf");
Time(&time2);

difftm=Difftime(&time1, &time2, 0);
sprintf(buf, "time is %.2f", difftm);
Log(buf);
```

SEE ALSO

Time, Eventtime, Paceconstant, Pacetne, Paceuniform

FUNCTION **Endfunction**

SYNTAX

```
Endfunction(str)
char *str;
```

DESCRIPTION

Parameters

`str` The name of the function. This parameter is a null-terminated string.

Comments

`Endfunction()` is used to mark the end of a function. This function is inserted into the script automatically during Capture when you specify the end of a function or, it can be inserted when editing your script.

`Endfunction()` works with and must have a corresponding `Beginfunction()` to define a task you wish to measure. The `Endfunction()` statement and its corresponding `Beginfunction()` must use the same function name, i.e. `str` parameter.

`Endfunction()` records the name of the function and the time at which the event occurred in the log file.

RETURN VALUE (not applicable)

EXAMPLES In the following example, the `Beginfunction()` and `Endfunction()` statements are placed in a C language function that consists of opening a window. The name of the function is `openwindow`. A function call is placed within the scenario and the actual function is listed after `Endscenario()`:

```
Empower();
{
AppWait(0.33);
WindowRcv("SfSfSfPt");

...

openwindow();

LeftButtonDown(132,91);
LeftButtonUp(158,212);


AppWait(3.52);
WindowRcv("ScSfSfSfPt");

Endscenario("example");
}

openwindow()
{
Beginsource("example");
Beginfunction("openwindow");

LeftDblPress(438,304);
LeftDblClick(438,304);


AppWait(0.24);
WindowRcv("PtCoCwCwDwAcSfCwCoSfCwCwCwCwCwCwCwCwSfAcPtPtPtPtPt");
WindowRcv("Pt");

CurrentWindow("New...");

/* Clicked (Button) (Cancel) */
LeftButtonDown(342,256);


Endfunction("openwindow");
Endsource();
}
```

0969794.102600

Execution of `Endfunction("query1")` could cause the following time stamp to be recorded in the script's log file.

```
>>>  26 Endfunction("query1") 04:11:23.29
```

SEE ALSO Beginfunction

FUNCTION **Endscenario**

SYNTAX

```
Endscenario(str)
char *str;
```

DESCRIPTION

Parameters

`str` The name of the script as defined in the Capture session. This name is a null-terminated character string.

Comments

This function is inserted automatically into your script file during Capture to define the end of a scenario. `Endscenario()` works with and must have a corresponding `Beginscenario()` to define a scenario. The `Endscenario()` function and its corresponding `Beginscenario()` must use the same scenario name, i.e., `str` parameter.

`Endscenario()` will record in the log file the name of the scenario and the time at which the event occurred.

EMPOWER/CS provides summary performance information for each scenario. For example, the Report tool provides scenario start and stop times, duration, throughput, and average response times.

By default, the duration of the test is determined by the time stamps of the first `Beginscenario()` and of the last `Endscenario()`.

RETURN VALUE (not applicable)

EXAMPLES The name of the scenario is taken from the name of the script source file. Initiation of capturing the script `example` would cause the following functions to be inserted at the beginning and end of the script:

```
Beginscenario("example")
...
Endscenario("example")
```

The log file created from script execution will contain time stamps for the beginning and end of the scenario.

Example:

```
>>>  10 Beginscenario("example") 12:05:29.00
...
>>>  462 Endscenario("example") 12:08:23.07
```

SEE ALSO Beginscenario

FUNCTION **Endsource**

SYNTAX `Endsource();`

DESCRIPTION

If a script is compiled with multiple object files, each object file's source script should include `Beginsource()` and `Endsource()` statements which are used to specify a source file. The `Beginsource()` function's parameter is a character string naming the source file; `Endsource()` has no parameters.

You should insert the `Beginsource()` function during script editing at the source file's entry point, typically just before the first executable statement in each function. The `Endsource()` function should be placed at the source file's exit point, typically just after the last executable statement in each function.

RETURN VALUE (not applicable)

EXAMPLES

In the following example, elements of a function called `logoff1()` may be contained in a separate script file called `exitapp.c`, as shown:

```
logoff1()
{
Beginsource("exitapp");
Beginfunction("logoff1");

Think(9.05);

LeftButtonDown(207,97);
LeftButtonUp(226,241);


AppWait(0.33);
WindowRcv("ScDwAc");
```

(continued on following page . . .)

```
CurrentWindow("Capture - script1",21,690,57,726);

Commit(LOG1);


Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endfunction("logoff1");
Endsource();
}
```

SEE ALSO Beginsource

FUNCTION **Endtimer**

SYNTAX

```
Endtimer(str)
char *str;
```

DESCRIPTION

Parameters

`str` A string that specifies the name of the timer to end

Comments

`Endtimer()` is used in a script for measuring specific script activity. These functions are time stamped in the executed script's log file and are used by Report for measuring response time of the specified activity. `Endtimer()` must be called after its corresponding `Begintimer()` function and the corresponding functions must have the same `str` parameter.

The `Begintimer()` and `Endtimer()` functions can be nested.

If the Capture option **Insert timer** is selected, the `Begintimer()` and `Endtimer()` functions are inserted automatically into the script by EMPOWER/CS to measure response time of database traffic. These functions are inserted around database traffic that occurs between two user events. A user event such as pressing the Return key on the keyboard or activating a pushbutton on screen may initiate database activity. The `str` parameter identifies the user event that initiated the database activity. For example, if activating a pushbutton initiated database activity, the parameter to the associated `Begintimer()` and `Endtimer()` functions would list a two-level tree structure that lists the parent window and the button.

The `Endtimer()` function will occur at the end of the database activity when a window is drawn on screen or another set of user input begins.

These functions also can be user-specified. Similar to the functions `Beginfunction()` and `Endfunction()`, they can be inserted manually

into the script when editing or by activating the EMPOWER/CS **Timer** button in the **Capture** window during Capture. If you select the **Timer** button, `Begintimer()` is inserted into the script measuring activity until the **End** button is selected to insert `Endtimer()`.

When editing your script, you may change the `str` parameter to a string that is more meaningful for your emulation.

RETURN VALUE (not applicable)

EXAMPLES In the following example, `Begintimer()` and `Endtimer()` were inserted into the script around database activity. Notice that the parameter to these functions lists the last user event, which was pressing the **OK** button in the window titled **Status**.

```
CurrentWindow("Status",240,180,408,301);
ButtonPush("OK",322,267);

WindowRcv("SfAcSfDw");

...

Begintimer("Status_OK");

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endtimer("Status_OK");
```

SEE ALSO Begintimer

FUNCTION **Eventtime**

SYNTAX

```
int Eventtime(event, p)
int event;
struct timevalue *p;
```

DESCRIPTION

Parameters

event	The name of an EMPOWER/CS event
p	A timevalue structure where the returned event time is stored

Comments

You can insert this function when editing your script file. `Eventtime()` retrieves the time at which the last of several events occurred. Valid events defined in `$EMPOWER/h/empower.h` are:

EVENT	DESCRIPTION
TSTART	time that script starts
TBF	time at last beginfunction
TEF	time at last endfunction
TBS	time at last beginscenario
TES	time at last endscenario
TBT	time at last begintimer
TET	time at last endtimer

The returned event time is stored in a timevalue structure pointed to by `p`. The `struct timevalue` is defined in `$EMPOWER/h/empowercs.h` as:

```
struct timevalue {
      long  sec;    /*seconds since Jan. 1 1970*/
      short hsec;   /*hundredths of a second*/
}
```

RETURN VALUE

`Eventtime()` returns `0` if successful and returns `-1` if the event is undefined.

EXAMPLES

This example script segment uses `Eventtime()` and `Difftime()` to record the duration of events in the log file. Before the EMPOWER Pace functions were developed, these functions helped the user to maintain a constant transaction throughput. (Some script content is left out for brevity.)

```
char buf[30];
double difftm;
struct timevalue time1, time2, time3, time4;

Beginscenario("script1");

....

Beginfunction("logout1");

...

Endfunction("logout1");

Endscenario("script1");

Eventtime(TBF, &time1);
Eventtime(TEF, &time2);
Eventtime(TBS, &time3);
Eventtime(TES, &time4);

difftm=Difftime(&time1,&time2,0);
sprintf(buf, "logout1 function was %.2f seconds", difftm);
Log(buf);

difftm=Difftime(&time3,&time4,0);
sprintf(buf, "scenario duration was %.2f seconds", difftm);
Log(buf);
```

The relevant portion of the script's log file follows:

```
>>>    20 Beginscenario("script1") 14:10:44.26
...
>>>   368 Beginfunction("logout1") 14:11:48.20
...
>>>   390 Endfunction("logout1") 14:11:54.37
>>>   352 Endscenario("script1") 14:11:54.37
>>>   359 Log("logout1 function was 6.17 seconds")
>>>   362 Log("scenario duration was 70.11 seconds")
```

SEE ALSO Difftime, Paceconstant, Pacetne, Paceuniform, Time

FUNCTION **Exec**

SYNTAX

```
Exec(curnum)
int curnum;
```

DESCRIPTION

Parameters

`curnum` An identifier of the cursor communication structure specified in the associated `Parse()`

Comments

This function is inserted into the script when the `Parse()` statement is executed on the SUT. During script execution, `Exec()` instructs the SUT to execute the current `Parse()` statement. `Exec()` also forces all data and other relevant information to be passed from the client to the SUT.

`Exec()` is called after a `Parse()` and all the `Bind()` or `Define()` statements associated with a specific cursor. For database queries, the requested rows are fetched with the `Fetch()` function after `Exec()` has been called.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the application and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES In the following script segment the `Exec()` statement is called for the cursor, `CUR1`:

```
Open(LOG1,CUR1);

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
Define(CUR1, "1", STRING, 5);
Define(CUR1, "2", STRING, 11);
Define(CUR1, "3", STRING, 10);
Define(CUR1, "4", STRING, 5);
Define(CUR1, "5", STRING, 10);
Define(CUR1, "6", STRING, 9);
Define(CUR1, "7", STRING, 9);
Define(CUR1, "8", STRING, 3);
Exec(CUR1);

Fetch(CUR1);
```

SEE ALSO Parse, Fetch

FUNCTION **Fetch**

SYNTAX

```
Fetch(curnum)
int curnum;
```

DESCRIPTION

Parameters

`curnum` An identifier of the cursor communication structure specified in the associated `Parse()`

Comments

This function is inserted into the script when the requested data in the associated `Parse()`, or SQL, statement is retrieved from the database. During script execution, `Fetch()` retrieves the data that satisfies the query. This data is retrieved from the database into a buffer on the UNIX script driver.

The number of rows of data to be fetched is specified in the `FETCHSIZE` option of the `Dbset()` function in the following format:

```
Dbset(curnum, FETCHSIZE, n)
```

The parameter `n` in this `Dbset()` function specifies the number of rows of data to be fetched with 1 as the default of `n`.

Each `Fetch()` statement returns the set of rows from the database that satisfies a query. After the last row has been returned, the next fetch will return an error that no remaining rows could be fetched. The `GetNextRow()` function is used to retrieve individual rows from the buffer on the UNIX script driver.

The `Fetch()` statement is called after the `Parse()` and `Exec()` functions.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to

edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the application and SUT and therefore, break the script during execution.

RETURN VALUE

`Fetch()` returns the number of rows fetched from the database. If no more rows can be fetched, an error will be returned.

EXAMPLES

In the following example, the `FETCHSIZE` for the cursor, `CUR1`, is set to 4 in the `Dbset()` function:

```
Dbset(CUR1, FETCHSIZE 4)
```

In the following script segment, `Fetch()` is called for the cursor, `CUR1`:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1,MAXARRSIZE,  64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 64);
Fetch(CUR1);
```

SEE ALSO Dbset, FetchRaw, GetNextRow, Parse

FUNCTION **FetchRaw**

SYNTAX

```
FetchRaw(curnum, pos, type, length)
int curnum, pos, type, length;
```

DESCRIPTION

Parameters

`curnum` An identifier of a cursor communication structure of the associated `Parse()`

`pos` The position of the specified variable in the SQL statement

`type` The variable's data type

`length` The number of bytes of data to retrieve from the SUT

Comments

The `FetchRaw()` function is inserted into the script instead of a `Fetch()` if the specified data to be retrieved is in binary form. This data is fetched in `length`-sized portions from the SUT.

The parameters to `FetchRaw()` specify the cursor number of the associated SQL statement, the position of the associated select-list item in the SQL statement, the data type of the output variable, and the number of bytes of data to retrieve from the database.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE `FetchRaw()` returns the number of rows fetched from the database. If no more rows can be fetched, an error will be returned.

EXAMPLES

The following example demonstrates a FetchRaw() function that fetches binary data in 1024 byte sections until no more data can be fetched:

```
long n;
do {
   n=FetchRaw(CUR1,1,BINARY,1024);
} while(n>0)'
```

SEE ALSO

Fetch

File Input/Output Functions

You can insert EMPOWER/CS File Input/Output functions into your script when editing. These functions are used to read and write files. Such capabilities are useful for load tests requiring interaction with data files on the UNIX driver machine and for simplifying complex scripts such as database entry scripts.

The EMPOWER/CS file input/output functions are used in your scripts to read data from a file, send data from the file to the SUT, receive data from the SUT, and write those data to a file. These functions simplify the C language statements that would need to be added to scripts to accomplish the same thing.

The environment variable `E_FIOPATH` can be used to specify the directory in which the files to be accessed reside. A file must be opened before it can be accessed with the file input/output functions. If a file contains NULL characters, an error will occur when the file is read by an input/output function.

Three global variables are used for file input/output. They are defined automatically as follows:

```
unsigned char *FIOBUFFER
int FIOLEN
int FIOBUFFERSZ
```

The variable `FIOBUFFER` is a pointer to the characters read from the file. This variable often is used when sending data read from a file to the SUT. The variable `FIOLEN` is the number of valid characters in `FIOBUFFER`. If the value of `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file (EOF) was reached. The variable `FIOBUFFERSZ` is the maximum size of the data that can be read at one time. The default value of `FIOBUFFERSZ` is 512 characters. If the value of `FIOBUFFERSZ` is redefined in a script, it must be redefined before any file input/output functions that reference the file are encountered.

These functions are described in the following reference entries.

FUNCTION **Fioautorewind**

SYNTAX

```
Fioautorewind(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` The file used for the operation

Comments

The `Fioautorewind()` function automatically rewinds the file pointer to the beginning of the file specified in `filename` whenever an end-of-file is encountered. If the end-of-file is not encountered, scripts will bypass this function. `Fioautorewind()` is useful if multiple scripts read data from one file.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following examples show you how to use the rewind functions to re-use data in an input file. Notice that `Fioautorewind()` saves you some programming time because you do not have determine if you are at the end of the file:

```
Fioautorewind("info");
Fioreadline("info");
Type("%s", FIOBUFFER);
```

OR

```
Fioreadline("info");
if (FIOLEN==-1){
Fiorewind("info");
Fioreadline("info");
}
Type("%s", FIOBUFFER);
```

SEE ALSO Fiorewind

FUNCTION **Fioclose**

SYNTAX

```
Fioclose(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` -	The file to close

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fioclose()` function closes the file in the parameter `filename`. Open files automatically are closed when the script exits. However, for good programming practice, using this function in your script file may be useful.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following script opens the "`names`" file, reads a line from the file, and transmits the name to the SUT with the `Type()` function. Then it closes the "`names`" file, opens the "`numbers`" file, reads a line from this file, and transmits the number to the SUT with the `Type()` function. The "`numbers`" file also is closed with `Fioclose()`.

```
Fioopen("names", "r");
Fioreadline("names");
Type("%s", FIOBUFFER);
Fioclose("names");

LeftButtonPress(360,211);


AppWait(0.06);
WindowRcv("SfSfPt");

Fioopen("numbers", "r");
Fioreadline("numbers");
Type("%s", FIOBUFFER);
Fioclose("numbers");
```

SEE ALSO Fioopen

0969794.102600

FUNCTION **Fiodelimiter**

SYNTAX

```
Fiodelimiter(filename, delimiters)
char *filename;
unsigned char *delimiters;
```

DESCRIPTION

Parameters

`filename`	The file for the operation
`delimiters`	The field delimiters for the specified file

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fiodelimiter()` function defines the field delimiters for the file `filename`. The default is "`\t\n`" where "`\n`" is always a delimiter ("`\t`" is tab and "`\n`" is new line or linefeed).

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example illustrates using the functions `Fioreadfield()` and `Fioreadfields()` to read one or more fields from a file. `Fiodelimiter()` specifies that the field delimiter in the file `inputfile` is a comma. (Some script content was left out for brevity.)

```
char last[20];
char first[20];

...

Fioopen("inputfile", "r");
Fiodelimiter("inputfile", ",");
Fioreadfield("inputfile");
Type("%s", FIOBUFFER);

LeftButtonPress(360,211);


AppWait(0.06);
WindowRcv("SfSfPt");

Fioreadfields("inputfile", 2, last, first);
Type("%s", last);

LeftButtonPress(357,252);


AppWait(0.06);
WindowRcv("SfSfPt");

Type("%s", first);
```

The following is a portion of the file inputfile:

```
312890463
doe,jane
294028190
smith, john
```

SEE ALSO Fioreadfield, Fioreadfields, Fioskipfield

FUNCTION **Fioopen**

SYNTAX

```
Fioopen(filename, mode)
char *filename, *mode;
```

DESCRIPTION

Parameters

`filename` The file to be opened

`mode` The mode for opening the file

Comments

See the description under File Input/Output functions for a general explanation of these functions.

The `Fioopen()` function opens the file `filename`. The parameter `mode` specifies how the file is opened. The following list demonstrates how the file is opened by specifying different `mode` parameters:

Mode	How File Is Opened
`r`	Opened at the beginning for reading only
`w`	Truncated or created for writing only
`a`	Opened at the end for writing only
`r+`	Opened at the beginning for reading and writing
`w+`	Truncated or created for reading or writing
`a+`	Opened at the end for reading and writing

Most of the File Input/Output functions will open the file automatically if it has not been opened previously with `Fioopen()`. If you plan to write to a file, be sure to call `Fioopen()` with the appropriate mode.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following script opens the "`names`" file, reads a line from the file, and transmits the name to the SUT with the `Type()` function. Then it closes the "`names`" file, opens the "`numbers`" file, reads a line from this

file, and transmits the number to the SUT with the `Type()` function. The "`numbers`" file is also closed with `Fioclose()`.

```
Fioopen("names", "r");
Fioreadline("names");
Type("%s", FIOBUFFER);
Fioclose("names");

LeftButtonPress(360,211);


AppWait(0.06);
WindowRcv("SfSfPt");

Fioopen("numbers", "r");
Fioreadline("numbers");
Type("%s", FIOBUFFER);
Fioclose("numbers");
```

SEE ALSO Fioclose

FUNCTION **Fioreadchar**

SYNTAX

```
Fioreadchar(filename, n)
char *filename;
int n;
```

DESCRIPTION

Parameters

`filename` The file for the operation

`n` The number of bytes to read from the specified file

Comments

See the description under File Input/Output functions for a general explanation of these functions.

The `Fioreadchar()` function reads `n` bytes from the file `filename`. If the file is not currently open, it is opened by `Fioreadchar`. A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

RETURN VALUE This function returns a pointer to the global variable `FIOBUFFER`.

EXAMPLES The following statements are used to read 10 characters from the "`letters`" file and send them to the SUT:

```
Fioreadchar("letters", 10);
Type("%s", FIOBUFFER);
```

SEE ALSO Fioclose, Fioopen, Fioreadfield, Fioreadfields, Fioreadline

0095759494.1026.00

FUNCTION **Fioreadfield**

SYNTAX

```
Fioreadfield(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

Comments

See the description under File Input/Output functions for a general explanation of these fucntions.

The `Fioreadfield()` function reads the next field from the file `filename` into `FIOBUFFER`. The fields are separated by the delimiter. If the file is not currently open, it is opened automatically. A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

RETURN VALUE This function returns a pointer to the global variable `FIOBUFFER`.

EXAMPLES The following example illustrates using the functions `Fioreadfield()` and `Fioreadfields()` to read one or more fields from a file. `Fiodelimiter()` is used to specify that the field delimiter in the file `inputfile` is a comma. (Some script content was left out for brevity.)

```
char last[20];
char first[20];

....

Fioopen("inputfile", "r");
Fiodelimiter("inputfile", ",");
Fioreadfield("inputfile");
Type("%s", FIOBUFFER);

LeftButtonPress(360,211);


AppWait(0.06);
WindowRcv("SfSfPt");

Fioreadfields("inputfile", 2, last, first);
Type("%s", last);

LeftButtonPress(357,252);


AppWait(0.06);
WindowRcv("SfSfPt");

Type("%s", first);
```

The following is a portion of the file inputfile:

```
312890463
doe,jane
294028190
smith, john
```

SEE ALSO Fioclose, Fiodelimiter, Fioopen, Fioreadchar, Fioreadfields, Fioreadline

FUNCTION **Fioreadfields**

SYNTAX

```
Fioeadfields(filename, n, arg0, ..., arg[n])
char *filename;
int n;
char *arg[n]
```

DESCRIPTION

Parameters

`filename`	The file to be used for the operation
`n`	The number of fields to read
`args`	The arguments where read fields are copied

Comments

This function reads `n` fields from the file `filename` as delimited by field delimiters. The default is `" \t\n"`, where "`\t`" is a tab and `"\n"` is always a delimiter. The fields are copied into arguments. If the file specified in `filename` is not currently opened, this function automatically opens it for reading.

RETURN VALUE

A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

EXAMPLES

The following example illustrates using the functions `Fioreadfield()` and `Fioreadfields()` to read one or more fields from a file. `Fiodelimiter()` is used to specify that the field delimiter in the file `inputfile` is a comma. (Some script content was left out for brevity.)

```
char last[20];
char first[20];

...

Fioopen("inputfile", "r");
Fiodelimiter("inputfile", ",");
Fioreadfield("inputfile");
Type("%s", FIOBUFFER);

LeftButtonPress(360,211);


AppWait(0.06);
WindowRcv("SfSfPt");

Fioreadfields("inputfile", 2, last, first);
Type("%s", last);

LeftButtonPress(357,252);


AppWait(0.06);
WindowRcv("SfSfPt");

Type("%s", first);
```

The following is a portion of the file inputfile:

```
312890463
doe,jane
294028190
smith, john
```

SEE ALSO Fioclose, Fiodelimiter, Fioopen, Fioreadchar, Fioreadfield, Fioreadline

FUNCTION **Fioreadline**

SYNTAX

```
Fioreadline(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fioreadline()` function reads the next line from the file `filename` into `FIOBUFFER`. If the file is not currently open, it is opened automatically. A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

RETURN VALUE This function returns a pointer to the global variable `FIOBUFFER`.

EXAMPLES The following script opens the "`names`" file, reads a line from the file, and transmits the name to the SUT with the `Type()` function. Then it closes the "`names`" file, opens the "`numbers`" file, reads a line from this file, and transmits the number to the SUT with the `Type()` function. The "`numbers`" file is also closed with `Fioclose()`.

```
Fioopen("names", "r");
Fioreadline("names");
Type("%s", FIOBUFFER);
Fioclose("names");

LeftButtonPress(360,211);
```

(continued on following page . . .)

```
AppWait(0.06);
WindowRcv("SfSfPt");

Fioopen("numbers", "r");
Fioreadline("numbers");
Type("%s", FIOBUFFER);
Fioclose("numbers");
```

SEE ALSO Fioclose, Fioopen, Fioreadchar, Fioreadfield, Fioreadfields

FUNCTION **Fiorewind**

SYNTAX

```
Fiorewind(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fiorewind()` function rewinds the file pointer to the beginning of the file specified in `filename`.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following examples show you how to use the rewind functions to reuse data in an input file. Notice that `Fioautorewind()` saves you some programming because you do not have to check to see if you are at the end of the file:

```
Fioautorewind("info");
Fioreadline("info");
Type("%s", FIOBUFFER);
```

OR

```
Fioreadline("info");
if (FIOLEN==-1){
Fiorewind("info");
Fioreadline("info");
}
Type("%s", FIOBUFFER);
```

SEE ALSO Fioautorewind, Fioclose, Fioseek

FUNCTION **Fioseek**

SYNTAX

```
Fioseek(filename, offset)
char *filename;
long offset;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

`offset` The offset of bytes from the beginning of the file

Comments

See the description under File Input/Output functions for a general explanation of these functions.

The `Fioseek()` function sets the file pointer to a specific byte in the file. The next byte read or written will occur at `offset` bytes from the beginning of the file. If the value of `offset` is equal to `FIOEND`, the seek will continue to the end of the file.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioautorewind, Fioclose, Fiorewind

FUNCTION **Fioshare**

SYNTAX

```
Fioshare(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` The file to be shared

Comments

See the description under File Input/Output functions for a general explanation of these functions.

The `Fioshare()` function identifies a file that is to be shared. It must be called before any other File I/O functions are called to reference the same file. `Fioshare` in a script presumes execution of the `fioshare` command at the UNIX script driver's shell prompt. The `fioshare` command creates a global variable that contains the offset for the next byte to be read from a shared file.

The value of the variable (offset) remains between tests, so you can continue to read an input file from the point left by the previous test. This saving of the offset is useful in tests that corrupt a database on the server. The ability to avoid the same transactions means you can avoid restoring the database before every test. You must execute the `fioshare` command if you want to resume reading from the beginning of the input file. For this reason, `fioshare` often is run from Mix command files that set up for a new test.

If your file to be shared includes database fields, you must be sure to use the `Gv_protect()` function to protect the database fields.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES Assume we require scripts to read commands from an input file called

cmds. The following command will create the global offset for the file:

```
$ fioshare  cmds
```

A segment in the script to read from cmds might look like the following. Each instance of this script will read different lines in the cmds file:

```
Fioshare("cmds");

Fioreadline("cmds");
if (FIOLEN == -1)
{
  Log("end of the date file");
  exit(1);
}
Type(FIOBUFFER, "^M", "");
}
```

The global offset is stored in a Global Variable. This Global Variable's name is the inode of the shared file which can be confirmed by typing the UNIX ls -i command with an argument of the shared file and then by executing the gv_stat command. This command lists the name, status, and value of EMPOWER/CS global variables. For example:

```
$ fioshare  cmds
$ ls  -i  cmds
59449 cmds
$ gv_stat
gv_stat:  EMPOWER/GV V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Name        Type             Value Allocated   Protector
----------  ---------------- ----- ---------   -------------
59449       long int             0         0
```

SEE ALSO   Fioclose, Fiounshare, gv_stat

FUNCTION **Fioskipchar**

SYNTAX

```
Fioskipchar(filename, n)
char *filename;
int n;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

`n` The number of characters to skip forward in the specified file

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fioskipchar()` function skips forward `n` characters in the file `filename`. If the file is not currently open, it is opened automatically by `Fioskipchar()`. The variables `FIOBUFFER` and `FIOLEN` are updated with the last characters read (the number of characters used to update `FIOBUFFER` and `FIOLEN` is defined by the variable `FIOBUFFERSZ`).

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioclose, Fioopen, Fioskipfield, Fioskipline

FUNCTION **Fioskipfield**

SYNTAX

```
Fioskipfield(filename, n)
char *filename;
int n;
```

DESCRIPTION

Parameters

`filename` The file to be used for the operation

`n` The number of fields to skip forward in the specified file

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fioskipfield()` function skips forward `n` fields in the file `filename`. The fields are separated by the delimiter. If the file is not currently open, it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last field read.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioclose, Fiodelimiter, Fioopen, Fioskipchar, Fioskipline

FUNCTION **Fioskipline**

SYNTAX

```
Fioskipline(filename, n)
char *filename;
int n;
```

DESCRIPTION

Parameters

`filename`	The file to be used for the operation
`n`	The number of lines to skip forward in the specified file

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fioskipline()` function skips forward `n` lines in the file `filename`. If the file is not currently open, it is opened automatically with `Fioskipline()`. The variables `FIOBUFFER` and `FIOLEN` are updated with the last line read.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioclose, Fioopen, Fioskipchar, Fioskipfield

FUNCTION **Fiounshare**

SYNTAX

```
Fiounshare(filename)
char *filename;
```

DESCRIPTION

Parameters

`filename` - The file to be used for the operation

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fiounshare()` function and the fiounshare shell command discontinue the sharing of a file. Neither the function nor the command remove the global variable offset for the file. They simply mark the global variable as being unusable for further Input/Output functions.

The `Fiounshare` function disables the sharing of the file offset only for the script that executes the function, and the `fiounshare` shell command disables the sharing of the file offset for all scripts currently sharing the file.

To remove the global variable offset, you must use the `gv_rm` shell command to remove the variable.

For example:

```
$ gv_rm -f 59449
```

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioclose, Fioshare, gv_rm, gv_stat

FUNCTION **Fiowritechar**

SYNTAX

```
Fiowritechar(filename, buf, n)
char *filename, *buf;
long n;
```

DESCRIPTION

Parameters

`filename`	The file to be used for the operation
`buf`	The file buffer
`n`	The number of bytes from the buffer

Comments

See the description under File Input/Output Functions for a general explanation of these functions.

The `Fiowritechar()` function writes `n` bytes from the buffer, `buf`, to the file `filename`. If the file is not currently open, it automatically is created or truncated and opened for reading and writing.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Fioclose, Fioopen

FUNCTION **GetNextRow**

SYNTAX

```
GetNextRow(curnum)
int curnum;
```

DESCRIPTION

Parameters

`curnum` An identifier of a cursor communication structure

Comments

The `GetNextRow()` function is inserted into your script after a `Fetch()` function when the client application sorts through the fetched rows of data.

During script execution, when the number of rows of data specified in the `Dbset()` option `FETCHSIZE` are fetched from the database, the data is placed into output variables in a buffer file on your UNIX driver machine. `GetNextRow()` sorts through the rows one at a time after they are fetched onto the UNIX script driver. This function generally occurs in a loop and may be used to verify that the requested data was retrieved or locate a specific row of data

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE `GetNextRow()` returns a string representation of the row retrieved.

EXAMPLES The following example demonstrates the execution of a database query in a script file. The query was generated on the cursor, `CUR1`. The EMPOWER/CS function `GetNextRow()` was inserted after the `Fetch()`:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
Define(CUR1, "1", STRING, 5);
Define(CUR1, "2", STRING, 11);
Define(CUR1, "3", STRING, 10);
Define(CUR1, "4", STRING, 5);
Define(CUR1, "5", STRING, 10);
Define(CUR1, "6", STRING, 9);
Define(CUR1, "7", STRING, 9);
Define(CUR1, "8", STRING, 3);
Exec(CUR1);

Fetch(CUR1);
GetNextRow(CUR1);
```

SEE ALSO Fetch

FUNCTION **GetIntVar**

SYNTAX

```
GetIntVar(curnum, var)
int curnum;
char *var;
```

DESCRIPTION

Parameters

`curnum` A cursor communication structure

`var` The name of the variable listed in the `Parse()` statement

Comments

You can insert `GetIntVar()` into your script file to return the current value of the specified variable that was listed in the `Parse()` statement.

This function should be inserted into the script after the `Fetch()` and `GetNextRow()` functions.

RETURN VALUE This function returns an integer for the current value of the variable.

EXAMPLES The following example demonstrates using `GetIntVar()` within a script:

```
int empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);
```

(continued on the following page . . .)

```
Dbset(CUR1, FETCHSIZE, 1);
while (Fetch(CUR1) != 0){
  GetNextRow(CUR1);
  empno=GetIntVar(CUR1, "2");
  printf("empno is %d\n", empno);
}
```

SEE ALSO CmpVar, GetVar, SetIntVar, SetVar

FUNCTION **GetVar**

SYNTAX

```
GetVar(curnum, var)
int curnum;
char *var;
```

DESCRIPTION

Parameters

`curnum` A cursor communication structure

`var` The name of the variable

Comments

You can insert the `GetVar()` function into your script to return the current value of the variable specified in the `Parse()` statement (as either the name or position)

This function should be inserted after the `Fetch()` or `GetNextRow()` functions.

RETURN VALUE

This function returns a string for the current value of the specified variable.

EXAMPLES

The following example demonstrates using `GetVar()` within a script:

```
char *empname, *empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);
```

(continued on following page . . .)

```
Dbset(CUR1, FETCHSIZE, 1);
while (Fetch(CUR1) != 0){
  GetNextRow(CUR1);
  empname=GetVar(CUR1, "1");
  printf("empname is %s\n", empname);
  empno=GetVar(CUR1, "2");
  printf("empno is %d\n", *(int *)empno);
}
```

SEE ALSO GetIntVar, SetIntVar, SetVar

COMMAND gv_add

SYNTAX `gv_add name value`

DESCRIPTION The gv_add command is entered at the command line and updates the value of a specified variable by adding a specified amount to the variable's current value. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, suppose the variable `users` has a value of 5 and you wish to change the value by adding 4. The interaction would be as follows:

```
$ gv_add users 4
5
$ gv_read users
9
```

SEE ALSO Gv_add

FUNCTION **Gv_add, Gv_addv**

SYNTAX

```
int Gv_add(name, value)
char *name;
int value;

int Gv_addv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The operand for the operation
`oldvalue`	The pointer location where the original value should be stored

Comments

The `Gv_add` function updates the value of a specified variable by adding a specified amount to the current value of a variable. You can insert these functions into your script when editing the script file.

`Gv_add()` is used if the variable is an integer, and `Gv_addv()` is used if the variable is not an integer.

RETURN VALUE

`Gv_add()` returns the original value of the specified variable. `Gv_addv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

The following example demonstrates using `Gv_add()` in a script file. In this example, if the variable `customer` is equal to zero, then 2 will be added to the current value of the variable `users`.

```
if (customer == 0)
  Gv_add(users, 2);
```

SEE ALSO gv_add

FUNCTION **Gv_alloc**

SYNTAX

```
void Gv_alloc(name, type)
char *name, *type;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`type`	The global variable type

Comments

The `Gv_alloc()` function allocates the script's access to the specified global variable. Access should be allocated for a variable so that a script can use the variable in subsequent Global Variable functions. The `Gv_alloc()` function should be the first function in the script if the script references a global variable.

An error will result if the `name` and `type` parameters of the `Gv_alloc()` function do not match the actual name and type of the variable specified when the variable was created with the `gv_init` command.

An error will result if the specified global variable does not exist, if the specified variable type does not match the global variable type, or if the global variable has already been allocated to the script.

If an error occurs, the script exits and an error message is sent to the standard error destination.

RETURN VALUE (not applicable)

EXAMPLES To allocate a script's access to the variable `users` which is an integer type, the following function must be included in the script file:

```
Gv_alloc("users", "int");
```

The variable `users` must be initialized at the shell prompt:

```
$ gv_init user int 50
```

SEE ALSO Gv_free, gv_init

COMMAND gv_and

SYNTAX `gv_and name value`

DESCRIPTION The `gv_and` command is entered at the shell prompt and updates the value of a specified variable by applying a bit-wise AND masking operation to the variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_and

FUNCTION **Gv_and, Gv_andv**

SYNTAX

```
int Gv_and(name, value)
char *name;
int value;

int Gv_andv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

`value` The operand for the operation

`oldvalue` The pointer location where the original value should be stored (`Gv_andv()`)

Comments

The `Gv_and()` and `Gv_andv()` functions update the value of a specified variable by applying a bit-wise AND masking operation to the variable. You can insert these functions into your script when editing.

`Gv_and()` is used if the variable is an integer and `Gv_andv()` is used if the variable is not an integer.

RETURN VALUE

`Gv_and()` returns the original value of the specified variable and `Gv_andv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_and

COMMAND **gv_dec**

SYNTAX `gv_dec name`

DESCRIPTION The `gv_dec` command is entered at the shell prompt and decreases the value of the specified variable by one. When the `gv_dec` command is entered, the original value is written to the standard output destination before the new, decremented value is assigned.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example the current value of the variable `count` is 5. To decrease this value by one use the following `gv_dec` command. Then, use the `gv_read` command to get the new value of the variable:

```
$ gv_dec  count
5
$ gv_read  count
4
```

SEE ALSO Gv_dec, Gv_inc, gv_inc

008597994.102600

FUNCTION **Gv_dec, Gv_decv**

SYNTAX

```
int Gv_dec(name)
char *name;

int Gv_decv(name, value)
char *name;
```

DESCRIPTION

Parameters

name — The name of the global variable

value — The pointer location where the original value should be stored (for Gv_decv())

Comments

The Gv_dec() and Gv_decv() functions update the value of a specified variable by subtracting one from the current value. Gv_dec() is used if the variable is an integer, and Gv_decv() is used if the variable is not an integer.

RETURN VALUE

The Gv_dec() function returns the original value of the specified variable and the Gv_decv() function copies the original value to a pointer location. After the value of the variable has been decremented, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

The following example shows that if the new value of global "amount" is zero then the script calls function "close_account". The script gets the new value of "amount" by subtracting one from the current value.

```
if(Gv_dec("amount") == 0)
  close_account();
```

SEE ALSO

Gv_inc, Gv_incv, gv_dec, gv_inc

COMMAND **gv_div**

SYNTAX `gv_div name value`

DESCRIPTION The `gv_div` command is entered at the shell prompt and updates the value of a specified variable by dividing the variable's current value by a specified amount. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, suppose the variable `users` has a current value of 8 and you wish to change the value by dividing it by 4. The interaction would be as follows:

```
$ gv_div  users  4
8
$ gv_read  users
2
```

SEE ALSO Gv_div

FUNCTION **Gv_div, Gv_divv**

SYNTAX

```
int Gv_div(name, value)
char *name;
int value;

int Gv_divv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The operand for the operation
`oldvalue`	The pointer location where the original value should be stored (for `Gv_divv()`)

Comments

The `Gv_div()` and `Gv_divv()` functions update the value of a specified variable by dividing the current value of the variable by a specified amount. You can insert this function into your script when editing the script file.

`Gv_div()` is used if the variable is an integer, and `Gv_divv()` is used if the variable is not an integer.

RETURN VALUE `Gv_div()` returns the original value of the specified variable and `Gv_divv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_div

FUNCTION **Gv_free**

SYNTAX

```
void Gv_free(name)
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

Comments

The `Gv_free()` function de-allocates a script's access to a specified variable. This function can be inserted into your script file when editing.

The script will not execute Global Variable functions for a variable that has been de-allocated unless access is re-allocated. If a script attempts to de-allocate a variable with `Gv_free()` and the variable has been protected by the script, the variable will be unprotected before the `Gv_free()` function completes.

Note: Using the `Gv_free()` function is optional. When a script exits, all allocated global variables will de-allocate automatically.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates that the script allocates and reads the global variable `amount`, then it de-allocates access to the global variable with `Gv_free()`.

```
int number;
...
Gv_alloc("amount");
number=Gv_read("amount");
if (number > 10)
  order();
Gv_free("amount");
```

SEE ALSO Gv_allocate

COMMAND **gv_getparallel**

SYNTAX `gv_getparallel name`

DESCRIPTION The `gv_getparallel` command is entered at the shell prompt and returns the current value of the specified parallel variable to the standard output destination.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES This example returns a value of 50 for the parallel variable `workers`:

```
$ gv_getparallel  workers
50
```

SEE ALSO Gv_getparallel, Gv_parallel, Gv_setparallel, Gv_unparallel, gv_parallel, gv_setparallel, gv_unparallel

FUNCTION **Gv_getparallel**

SYNTAX

```
unsigned short int Gv_getparallel(name);
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

Comments

The `Gv_getparallel()` function returns the current value of the specified parallel variable. This function can be inserted into your script file when editing.

RETURN VALUE

The `Gv_getparallel()` function returns the current value of the specified parallel variable. This value should be stored in a local variable. If the variable type read is not `int`, the variable's current value is returned cast as an integer. If an error occurs, the script will exit and an error message is sent to the standard error destination.

SEE ALSO

Gv_parallel, Gv_setparallel, Gv_unparallel, gv_getparallel, gv_parallel, gv_setparallel, gv_unparallel

08957954.102600

COMMAND **gv_inc**

SYNTAX `gv_inc name`

DESCRIPTION The `gv_inc` command is entered at the shell prompt and increases the value of the specified variable by one. When the `gv_inc` command is entered, the original value is written to the standard output destination before the new, incremented value is assigned.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, the current value of the variable `count` is 5. To increment the value by one use the following `gv_inc` command. Then, enter a `gv_read` command to get the new value:

```
$ gv_inc  count
5
$ gv_read  count
6
```

SEE ALSO Gv_dec, Gv_inc, gv_dec

FUNCTION **Gv_inc, Gv_incv**

SYNTAX

```
int Gv_inc(name)
char *name;

int Gv_incv(name, value)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The pointer location where the original value should be stored (for `Gv_incv()`)

Comments

You can insert these functions into your script file when editing.

The `Gv_inc()` and `Gv_incv()` functions update the value of the specified variable by adding one to the current value. `Gv_inc()` is used if the variable is an integer, and `Gv_incv()` is used if the variable is not an integer.

RETURN VALUE

The `Gv_inc()` function returns the original value of the specified variable and the `Gv_incv()` function copies the original value to a pointer location. After the value of the variable has been incremented, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

The following example can be used to increment the integer variable `count`:

```
Gv_alloc("count", "int");
Gv_inc("count");
```

To increment the non-integer variable `balance`, the following example can be used:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_incv("balance", &curbalance);
```

SEE ALSO Gv_dec, Gv_decv, gv_dec, gv_inc

COMMAND gv_init

SYNTAX

```
gv_init name [type] value
```

DESCRIPTION

The gv_init command is entered at the shell prompt and is used to initialize a variable. If the variable does not exist, it is created with the specified type and initial value. If the variable exists when the gv_init command is entered, the variable is reset to the specified value.

The parameter type is required if the variable does not exist. If the variable exists, the parameter type is optional and the type and value specified in gv_init must correspond to the existing variable type. If a different type is specified or if the new value specified does not correspond to the existing type, the command fails.

The default maximum number of variables is 128. The maximum length of a variable name is 14 characters. The maximum length of the value of a string variable is 32 characters.

RETURN CODE

If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES

The gv_init command creates a variable with a specified name, variable type, and initial value. In the following example, gv_init is the Global Variable command, customer is the variable name, int specifies that customer is an integer, and 100 is the initial value of the variable customer.

```
$ gv_init  customer  int  100
```

To specify the same variable as a parallel variable, you would enter the following:

```
$ gv_init  customer  parallel  100
```

Global variables generally are created at the UNIX driver machine with the `gv_init` command, then accessed in a script with the `Gv_alloc()` function:

To use the following variable `users` during script execution:

```
$ gv_init users int 0
```

The following function should be included in the script:

```
Gv_alloc("users", "int");
```

SEE ALSO Gv_alloc, gv_seg

0095794.102600

COMMAND gv_lshift

SYNTAX `gv_lshift name value`

DESCRIPTION The `gv_lshift` command is entered at the shell prompt and updates the value of a specified variable by performing a bit-wise shift to the left on a variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_lshift, Gv_rshift

FUNCTION **Gv_lshift, Gv_lshiftv**

SYNTAX

```
int Gv_lshift(name, value)
char *name;
int value;

int Gv_lshiftv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The operand for the operation
`oldvalue`	The pointer location where the original value should be stored (for `Gv_lshiftv()`)

Comments

These functions are used with the EMPOWER/GV tool and are inserted into your script file when editing.

The `Gv_lshift()` and `Gv_lshiftv()` functions update the value of a specified variable by performing a bit-wise shift to the left on the variable.

`Gv_lshift()` is used if the variable is an integer, and `Gv_lshiftv()` is used if the variable is not an integer.

RETURN VALUE `Gv_lshift()` returns the original value of the specified variable and `Gv_lshiftv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO Gv_rshift, Gv_rshiftv, gv_lshift, gv_rshift

COMMAND **gv_mod**

SYNTAX `gv_mod name value`

DESCRIPTION The gv_mod command is entered at the shell prompt and updates the value of a specified variable by performing a modulo operation on a variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, when `count=10` and the user performs a modulo operation with 3, the new value of `count` is 1.

```
$ gv_init count int 10
$ gv_mod count 3
10
$ gv_read count
1
```

When count=10 and the user performs a modulo operation with 2, the new value of count is 0.

```
$ gv_init count int 10
$ gv_mod count 2
10
$ gr_read count
0
```

SEE ALSO Gv_mod, Gv_modv

FUNCTION **Gv_mod, Gv_modv**

SYNTAX

```
int Gv_mod(name, value)
char *name;
int value;

int Gv_modv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The operand for the operation
`oldvalue`	The pointer location where the original value should be stored (for `Gv_modv()`)

Comments

The `Gv_mod()` and `Gv_modv()` functions update the value of a specified variable by performing a modulo operation on the variable. You can insert these functions into your script when editing your script file.

`Gv_mod()` is used if the variable is an integer, and `Gv_modv()` is used if the variable is not an integer.

RETURN VALUE

`Gv_mod()` returns the original value of the specified variable and `Gv_modv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_mod

COMMAND **gv_mul**

SYNTAX `gv_mul name value`

DESCRIPTION

The `gv_mul` command is entered at the shell promt and updates the value of a specified variable by multiplying the variable's current value by a specified amount. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE

If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES

In the following example the current value of the variable `users` is 5. To change the value of this variable by multiplying by 10, use the following `gv_mul` command. Then, enter the `gv_read` command to get the changed value:

```
$ gv_mul users 10
5
$ gv_read users
50
```

SEE ALSO Gv_mul, Gv_mulv

FUNCTION **Gv_mul, Gv_mulv**

SYNTAX

```
int Gv_mul(name, value)
char *name;
int value;

int Gv_mulv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

name The name of the global variable

value The operand for the operation

oldvalue The pointer location where the original value should be stored (for Gv_mulv())

Comments

The Gv_mul() and Gv_mulv() functions update the value of a specified variable by multiplying the current value of the variable by a specified amount. You can insert these functions into your script file when editing.

Gv_mul() is used if the variable is an integer, and Gv_mulv() is used if the variable is not an integer.

RETURN VALUE Gv_mul() returns the original value of the specified variable and Gv_mulv() copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES In the following example, the value of the variable count is multiplied by four. The value of count prior to the multiplication is stored in curcount:

```
int curcount;
Gv_alloc("count", "int");
curcount = Gv_mul("count", 4);
```

SEE ALSO gv_mul

COMMAND gv_or

SYNTAX `gv_or name value`

DESCRIPTION The `gv_or` command is entered at the command line and updates the value of a specified variable by applying a bit-wise OR masking operation to the variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_or

0959794.102600

FUNCTION Gv_or, Gv_orv

SYNTAX

```
int Gv_or(name, value)
char *name;
int value;

int Gv_orv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

name The name of the global variable

value The operand for the operation

oldvalue The pointer location where the original value should be stored (for Gv_orv())

Comments

The Gv_or() and Gv_orv() functions update the value of a specified variable by applying a bit-wise OR masking operation to the variable. These functions can be inserted in your script when editing the script file.

Gv_or() is used if the variable is an integer, and Gv_orv() is used if the variable is not an integer.

RETURN VALUE Gv_or() returns the original value of the specified variable and Gv_orv() copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_or

COMMAND gv_parallel

SYNTAX `gv_parallel name`

DESCRIPTION The `gv_parallel` command is entered at the shell prompt and waits for the value of the specified parallel variable to become greater than zero then decrements the variable by one. When used in conjunction with the `gv_unparallel` command, the parallel variable becomes an "on/off" switch to control multiple script execution.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_getparallel, Gv_parallel, Gv_setparallel, Gv_unparallel, gv_getparallel, gv_setparallel, gv_unparallel

FUNCTION Gv_parallel

SYNTAX

```
void Gv_parallel(name)
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

Comments

The `Gv_parallel()` function waits for the value of the specified parallel variable to become greater than zero then decrements the variable by one. When used in conjunction with the `Gv_unparallel()` function, the parallel variable acts as an "on/off" switch to control multiple script execution. You can insert this function into your script file when editing.

RETURN CODE

If the function succeeds, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

In this example, parallel Global Variables control execution of multiple scripts. Parallel variable commands and functions allow a subset of emulated users to execute a portion of the script at a given time. Specifically, during an emulation involving 500 users, a portion of the test may be executed in parallel by only 50 users.

Each user will execute one script. The Global Variable `worker` is created from the command line as a parallel type variable with an initial value of 50:

```
$ gv_init worker parallel 50
```

The following script is executed by all users (some script content is left out for brevity):

```
login()

{/* login transactions - not detailed here */}

Empower(argc, argv)
       int argc;
       char *argv[];
{
       Gv_alloc("worker", "parallel");
       login();
       Beginscenario("example");

       /* ... numerous transactions */

       Gv_parallel("worker");

       /* ... limited portion of transactions */

       Gv_unparallel("worker");

       /* ... numerous transactions */

       Endscenario("example");
}
```

The first 50 users to reach the `Gv_parallel()` function during script execution immediately will execute the limited portion of transactions. The remaining 450 users will reach the `Gv_parallel()` function and will wait.

As the 50th user executes the `Gv_parallel()` function, the value of the variable `worker` is decremented to zero. As each of the first 50 scripts reaches the `Gv_unparallel()` function, the value of `worker` is incremented to be greater than zero, so that one of the waiting scripts can execute the limited portion of transactions. This process ensures that only the first 50 scripts will execute the specified transactions at the same time.

SEE ALSO Gv_getparallel, Gv_setparallel, Gv_unparallel, gv_getparallel, gv_parallel, gv_setparallel, gv_unparallel

COMMAND **gv_protect**

SYNTAX `gv_protect [-f] name`

DESCRIPTION The `gv_protect` command is entered at the shell prompt and prevents scripts' access to a specified variable—only Global Variable Commands can access the variable. Scripts that try to access a protected variable will pause until the variable has been unprotected. The `-f` option forces protection of the variable even if the variable currently is protected by a script.

If a script has protected a variable and `gv_protect -f` is executed from the shell to protect the variable, the script will execute as though it no longer protects the variable. The script will block, waiting for the shell to unprotect the variable. The `Gv_unprotect()` function executed in the script has no effect. (*Note:* The `-f` option of `gv_protect` is rarely used.)

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination. If the `-f` option is used to force protection of a variable, a warning message will be returned.

EXAMPLES In this example, three scripts (`s1`, `s2`, and `s3`) are started and synchronized by accessing a variable `synch` which is protected at the command line with the `gv_protect` command. Then, the variable is unprotected. The `gv_stat` command is used to demonstrate the variable being protected and then unprotected. The interaction could look like the following:

```
$ gv_init synch int 0
$ gv_protect synch
$ gv_stat
Name               Type            Value Allocated Protector
------------------ --------------- ----- --------- ---------
synch              int                 0         0 CONSOLE
$ s1 &
[1] 3058
s1 ready
$ s2 &
[2] 3060
$ s3 &
[3] 3062
s2 ready
s3 ready
$ gv_stat
Name               Type            Value Allocated Protector
------------------ --------------- ----- --------- ---------
synch              int                 0         3 CONSOLE
$ gv_unprotect synch
$ gv_stat
Name               Type            Value Allocated Protector
------------------ --------------- ----- --------- ---------
synch              int                 3         3 NONE
```

SEE ALSO Gv_protect, Gv_unprotect, gv_unprotect

FUNCTION Gv_protect

SYNTAX

```
void Gv_protect(name)
char *name;
```

DESCRIPTION

Parameters

name The name of the global variable

Comments

The `Gv_protect()` function prevents access to the specified variable by other EMPOWER/CS scripts. Only the script executing the `Gv_protect()` function can access the variable. Other scripts trying to access a protected variable will pause until the variable has been unprotected. You can insert this function into your script when editing the script file.

An error will result if the `Gv_protect()` function is executed for a variable that does not exist or to which access has not been allocated. Also, an error will result if the script attempts to protect a variable which it already has protected.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates that the global variable `users` is protected, tested, and then unprotected.

```
Gv_alloc("users", "int");
Gv_protect("users");
if (Gv_test("users", "==", 3))
 Gv_write("users", 0);
Gv_unprotect("users");
Gv_inc("users");
```

SEE ALSO Gv_unprotect, gv_protect, gv_unprotect

COMMAND gv_read

SYNTAX `gv_read name`

DESCRIPTION The gv_read command is entered at the shell prompt and returns the current value of the specified variable to the standard output destination.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In this example the gv_read command returns a current value of 3 for the variable users:

```
$ gv_read  users
3
```

SEE ALSO Gv_read

FUNCTION **Gv_read, Gv_readv**

SYNTAX

```
int Gv_read(name)
char *name;

int Gv_readv(name, value)
char *name;
```

DESCRIPTION *Parameters*

`name`	The name of the global variable
`value`	Where the value of the global variable is stored after it was read (for `Gv_readv()`)

Comments

The `Gv_read()` function returns the current value of the specified variable. The `Gv_read()` function assumes the variable is an integer. If the variable is not an integer, the `Gv_readv()` function should be used. The `Gv_readv()` function reads the value specified by the parameter `name` and stores it in the variable specified by the parameter value.

RETURN VALUE The `Gv_read()` function returns the current value of the specified global variable. This value should be stored in a local variable. If the type of the variable read is not `int`, the current value of the variable is returned cast as an integer. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES If you want a script to read the current value of a global variable called `balance` and the value is an integer, use the `Gv_read()` function in the following script file entries. In this example, the `Gv_read()` function stores the current value of the variable `balance` in the variable `curbalance`:

```
int curbalance;
Gv_alloc("balance", "int");
curbalance = Gv_read("balance");
```

If the value of `balance` is not an integer, use the `Gv_readv()` function as in the following script file entries:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_readv("balance", &curbalance);
```

In the above example, the `Gv_readv()` function retrieves the current value of the variable `balance` and stores it in the floating point variable `curbalance`.

SEE ALSO gv_read

COMMAND gv_rm

SYNTAX `gv_rm [-f] [-r | name1 name2 ...]`

DESCRIPTION The gv_rm command is entered at the shell prompt and removes the specified variables from shared memory. If a variable currently is allocated to a script, the command will fail. If the -f option is used, each variable will be removed even if it is allocated to a script. If the -f option is used to remove a variable allocated to a script, a warning message will appear on screen and the variable will be removed. The script will exit the next time it attempts to use the variable. The -r option removes all global variables.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES The following example command will remove the variables users and transactions:

```
$ gv_rm users transactions
```

SEE ALSO gv_seg

COMMAND **gv_rshift**

SYNTAX `gv_rshift name value`

DESCRIPTION The `gv_rshift` command is entered at the shell prompt and updates the value of a specified variable by performing a bit-wise shift to the right on the variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_rshift, Gv_lshift, gv_lshift

FUNCTION Gv_rshift, Gv_rshiftv

SYNTAX

```
int Gv_rshift(name, value)
char *name;
int value;

int Gv_rshiftv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

name The name of the global variable

value The operand for the operation

oldvalue The pointer location where the original value should be stored (for `Gv_rshiftv()`)

Comments

The `Gv_rshift()` and `Gv_rshiftv()` functions update the value of a specified variable by performing a bit-wise shift to the right on the variable. You can insert these functions into your script when editing the script file.

`Gv_rshift()` is used if the variable is an integer, and `Gv_rshiftv()` is used if the variable is not an integer.

RETURN VALUE `Gv_rshift()` returns the original value of the specified variable and `Gv_rshiftv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO Gv_lshift, Gv_lshiftv

COMMAND **gv_seg**

SYNTAX `gv_seg [number-of-variables|-r]`

DESCRIPTION

The `gv_seg` command is entered at the shell prompt and creates a shared memory segment to support 128 global variables by default. When creating a new shared memory segment, the `number-of-variables` argument defines the number of variables the new segment will support. The `-r` option removes the existing shared memory segment which removes all existing global variables. If you want to create a new shared memory segment with a different size, the existing shared memory segment must be removed first.

RETURN CODE

If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES

To create a shared memory segment that supports 150 global variables, use the following command (assuming a shared memory segment does not already exist):

```
$ gv_seg  150
```

If a shared memory segment already exists, as would be the case if a Global Variables Command had already been executed, the existing shared memory segment must be removed before the new segment is created:

```
$ gv_seg  -r
$ gv_seg  150
```

SEE ALSO gv_init, gv_rm

COMMAND **gv_setparallel**

SYNTAX `gv_setparallel name value`

DESCRIPTION The `gv_setparallel` command is entered at the shell prompt and assigns a new value to the specified parallel variable. The new value is provided as the second parameter. When the `gv_setparallel` command is entered, the original value is written to the standard output destination before the new value is assigned.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, the parallel variable users has a current value of 50 and is reassigned a value of 60:

```
$ gv_setparallel  users  60
50
$ gv_getparallel  users
60
```

SEE ALSO Gv_getparallel, Gv_parallel, Gv_setparallel, Gv_unparallel, gv_getparallel, gv_parallel, gv_unparallel

FUNCTION **Gv_setparallel**

SYNTAX

```
void Gv_setparallel(name, value)
char *name;
unsigned short int value;
```

DESCRIPTION

Parameters

`name`	The name of the parallel global variable
`value`	The new value of the specified parallel variable

Comments

The `Gv_setparallel()` function assigns a new value to the specified parallel variable. The new value is listed as the second parameter. You can insert this function into your script when editing your script file.

RETURN CODE

If the function succeeds, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO

Gv_getparallel, Gv_parallel, Gv_unparallel, gv_getparallel, gv_parallel, gv_setparallel, gv_unparallel

COMMAND **gv_stat**

SYNTAX `gv_stat [name | -s]`

DESCRIPTION The `gv_stat` command is entered at the shell prompt and returns status information to the standard output destination for all variables or for a specified variable. The parameter is either a variable name or "`-s`". If the parameter provides the name of a variable, status information is provided for that variable only. If the `-s` option is specified, summary information is provided indicating the number of global variables that exist compared to the number of variables possible.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES The `gv_stat` command sends to the standard output destination a table showing the name, type, and value of the specified variable, with the number of scripts that have allocated the variable and the identity of the variable's protector, if applicable. For example:

```
$ gv_stat  users
Name            Type            Value          Allocated Protector
--------------- --------------- -------------- --------- ---------
users           int                          3         0 NONE
```

If gv_stat is executed without an argument, information will be listed for all variables:

```
$ gv_stat
Name            Type            Value          Allocated Protector
--------------- --------------- -------------- --------- ---------
users           int                          3         0 NONE
a               int                         -1         0 NONE
b               int                        100         0 NONE
```

An example using the `-s` option of `gv_stat` follows:

```
$ gv_stat  -s
10/20 global variables exist
```

SEE ALSO Gv_stat

FUNCTION **Gv_stat**

SYNTAX

```
int Gv_stat(name, info)
char *name;
struct gv_info *info;
```

DESCRIPTION

Parameters

`name` The name of the global variable

`info` The structure where status information for the specified global variable is stored

Comments

The `Gv_stat()` function copies the following global variable information into a structure specified in the second parameter: current value, name, type, owner name, owner process ID, and the number of scripts allocated to the variable. You can insert this function into your script when editing the script file.

The `gv_info` structure is declared in the `empowerGV.h` file and is included automatically in your script by the EMPOWER/CS tool Cscc. This structure is demonstrated below:

```
struct gv_info {
     int index; /* >= 0 indicates allocated to this script */
     int owner; /* >= 0 indicates pid of protecting owner */
     int users; /* number of users allocated */
     char name[SHMAXNAMELEN];
     char type[SHMAXTYPELEN];
     union val {
          int i;
          char c;
          .... .;
     } value; /* current variable value */
};
```

The specified global variable does not need to be allocated to the script before executing `Gv_stat()`. If the variable is not allocated to the script, the value of the `gv_info->index` field will be -1 and the `gv_info->value` field will be empty.

If the specified variable is protected by another script, the `gv_info->value` field will be empty. The `gv_info->owner` field will contain the process ID of the protecting script.

RETURN VALUE

The `Gv_stat()` function stores status information for the specified global variable in the specified structure. If this process is successful, the return code is set to one. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

The following example demonstrates using `Gv_stat()` in a script:

```
struct gv_info info;

if( Gv_stat("users", &info) ) {
   if ( strcmp("int", info.type) ) {
         fprintf(stderr, "wrong type for global variable
users: %s\n", info.type);
       Exit(-1);
   }
}
else
   Gv_alloc("users", "int", 0);

Beginscenario("manager");
...
Endscenario("manager");
```

SEE ALSO

gv_stat

COMMAND gv_sub

SYNTAX gv_sub name value

DESCRIPTION The gv_sub command is entered at the shell prompt and updates the value of a specified variable by subtracting the specified amount from the variable's current value. The parameter value is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, each of the GV commands returns the current value of the variable to the standard output destination before performing the specified operation. Suppose the variable users has a current value of 5 and you wish to make the value 6, then change the value by subtracting 4. The interaction would be as follows:

```
$ gv_write users 6
5
$ gv_sub users 4
5
$ gv_read users
2
```

SEE ALSO Gv_sub

FUNCTION **Gv_sub, Gv_subv**

SYNTAX

```
int Gv_sub(name, value)
char *name;
int value;

int Gv_subv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The operand for the operation
`oldvalue`	The pointer location where the original value should be stored (for `Gv_subv()`)

Comments

The `Gv_sub()` and `Gv_subv()` functions update the value of a specified variable by subtracting a specified amount from the current value of the variable. You can insert these functions into your script when editing the script file.

`Gv_sub()` is used if the variable is an integer, and `Gv_subv()` is used if the variable is not an integer. The parameter `value` is the operand for the operation, and the parameter `oldvalue` (for functions ending in "v") specifies the pointer location where the original value should be stored.

RETURN VALUE `Gv_sub()` returns the original value of the specified variable and `Gv_subv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_sub

COMMAND **gv_test**

SYNTAX `gv_test name relation-string [value]`

DESCRIPTION The `gv_test` command is entered at the shell prompt and compares the current value of the specified variable to the specified parameter `value` according to the test relation given in the parameter `relation-string`. If the relation string is a logical comparison ("`$`" or "`!`"), the `value` argument will be ignored.

RETURN CODE If the relational comparison is true, the return code is set to zero and a "1" is written to the standard output destination. If the relational comparison is false, the return code is set to one and a "0" is written to the standard output destination. If an error occurs, the return code is set to two and an error message is sent to the standard error destination.

EXAMPLES In the following example, the variable `users` will be tested to determine if its value is equal to 2:

```
$ gv_read users
2
$ gv_test users "==" 2
1
```

Notice that 1 is written to the standard output because the relational comparison was true.

The following example demonstrates both true and false relational comparisons:

```
$ gv_init a int 5
$ gv_test a "<" 1
0
$ gv_test a ">" 1
1
```

SEE ALSO Gv_test

FUNCTION **Gv_test**

SYNTAX

```
int Gv_test(name, op, value)
char *name;
char *op;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`op`	The relational test
`value`	The value to which the variable is compared

Comments

The `Gv_test()` function performs a relational comparison between the specified variable and the test value, according to the specified test relation. You can insert this function into your script when editing the script file.

The type of the global variable tested must match the type of the specified comparison value (the `value` parameter).

RETURN VALUE

If the relational test is true, the return value is set to one. If the test is false, the return value is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

In the following example, the function `Gv_test()` will test the variable `quitflag` to determine if its value is equal to 1. If the comparison is true, the script will exit:

```
Gv_alloc("quitflag", "int");
if (Gv_test("quitflag", "==", 1))
  Exit(1);
```

SEE ALSO gv_test

COMMAND **gv_unparallel**

SYNTAX `gv_unparallel name`

DESCRIPTION The `gv_unparallel` command is entered at the shell prompt and increments the value of the variable by one. When used in conjunction with `gv_parallel`, the parallel variable becomes an "on/off" switch to control multiple script execution.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_getparallel, Gv_parallel, Gv_setparallel, Gv_unparallel, gv_getparallel, gv_parallel, gv_setparallel

FUNCTION **Gv_unparallel**

SYNTAX

```
void Gv_unparallel(name)
char *name;
```

DESCRIPTION

Parameter

name The name of the global variable

Comments

The `Gv_unparallel()` function increments the value of the variable by one. When used in conjunction with the `Gv_parallel()` function, the parallel variable acts as an "on/off" switch to control multiple script execution.

You can insert this function into your script when editing the script file.

RETURN CODE If the function succeeds, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO Gv_getparallel, Gv_parallel, Gv_setparallel, gv_getparallel, gv_parallel, gv_setparallel, gv_unparallel

COMMAND **gv_unprotect**

SYNTAX `gv_unprotect [-f] name`

DESCRIPTION The `gv_unprotect` command is entered at the shell prompt and removes protection from the specified variable, allowing access to the variable by other users and scripts. Only the user that protected a variable may unprotect it, unless the `-f` option is used. The `-f` option forces removal of protection for the variable even if the variable was protected by another user or by a script. If the `-f` option is used, a warning message will be returned. This option typically is used when a script protecting a variable terminates abnormally.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_protect, Gv_unprotect, gv_protect

FUNCTION **Gv_unprotect**

SYNTAX

```
void Gv_unprotect(name)
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

Comments

The `Gv_unprotect()` function removes protection from a specified variable, allowing other scripts to access the variable. Only the script that protected a variable may unprotect it. A variable that has been protected by a script may be unprotected at the UNIX script driver with the `gv_unprotect` command if the `-f` option is used.

An error will result if the `Gv_unprotect()` function is executed for a variable which does not exist or to which access has not been allocated. Also, an error will result if the script attempts to unprotect a variable which the script has not protected.

You can insert this function into your script when editing the script file.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates that the global variable `users` is protected, tested, and then unprotected.

```
Gv_alloc("users", "int");
Gv_protect("users");
if (Gv_test("users", "==", 3))
  Gv_write("users", 0);
Gv_unprotect("users");
Gv_inc("users");
```

SEE ALSO Gv_protect, gv_protect, gv_unprotect

COMMAND **gv_waituntil**

SYNTAX `gv_waituntil name relation-string [value]`

DESCRIPTION The `gv_waituntil` command is entered at the shell prompt and compares the current value of the specified variable to the specified parameter `value` according to the test relation given in the parameter `relation-string`. Operations executing the command from the UNIX script driver or shell script will pause until the relational comparison is true. If the relation string is a logical comparison ("$" or "!"), the `value` argument will be ignored.

RETURN CODE When the relational comparison becomes true, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_waituntil, Gv_waitwhile, gv_waitwhile

FUNCTION **Gv_waituntil**

SYNTAX

```
void Gv_waituntil(name, op, value)
char *name;
char *op;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`op`	The test relation
`value`	The value that the variable is compared to

Comments

The `Gv_waituntil()` function compares the current value of the specified variable to the specified parameter `value` according to the test relation given in the parameter `op`. Script execution pauses until the relational comparison is true. You can insert this function into your script when editing the script file.

EMPOWER/CS executes the specified comparison immediately when the `Gv_waituntil` function is encountered, and again each time the value of the global variable has been updated. If the `Gv_waituntil()` function is executed for a global variable which the script has already protected, an error will result.

RETURN VALUE When the relational comparison becomes true, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES In the following example, the `Gv_waituntil()` function makes the script wait until the value of the variable `users` is equal to 128:

```
Gv_alloc("users", "int");
Gv_inc("users");
Gv_waituntil("users", "==", 128);
```

SEE ALSO Gv_waitwhile, gv_waituntil, gv_waitwhile

0095794.102600

COMMAND gv_waitwhile

SYNTAX `gv_waitwhile name relation-string [value]`

DESCRIPTION The gv_waitwhile command is entered at the shell prompt and compares the current value of the specified variable to the specified parameter value according to the test relation given in the parameter relation-string. Operations executing the command at the UNIX script driver or shell script will pause while the relational comparison is true. If the relation string is a logical comparison ("$" or "!"), the value argument will be ignored.

RETURN CODE When the relational comparison becomes false, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_waitwhile, Gv_waituntil, gv_waituntil

0095795464.102600

FUNCTION **Gv_waitwhile**

SYNTAX

```
void Gv_waitwhile(name, op, value)
char *name;
char *op;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`op`	The test relation used for the comparison
`value`	The value that the variable is compared to

Comments

The `Gv_waitwhile()` function compares the current value of the specified variable to the specified parameter `value` according to the test relation given in the parameter `op`. Script execution pauses while the relational comparison is true. You can insert this function into your script when editing the script file.

EMPOWER executes the specified comparison immediately when the `Gv_waitwhile()` function is encountered, and again each time the value of the global variable has been updated. If the `Gv_waitwhile()` function is executed for a global variable which the script has already protected, an error will result.

RETURN VALUE

When the relational comparison becomes true, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

In the following example, the `Gv_waitwhile()` function makes the script wait while the value of the variable `count` is less than 20:

```
Gv_alloc("count", "int");
Gv_inc("count");
Gv_waitwhile("count", "<", 20);
```

SEE ALSO Gv_waituntil, gv_waituntil, gv_waitwhile

COMMAND **gv_write**

SYNTAX `gv_write name value`

DESCRIPTION The `gv_write` command is entered at the shell prompt and assigns a new value to the specified variable. When the `gv_write` command is entered, the original value is written to the standard output destination before the new value is assigned.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, suppose the variable `users` has a current value of 5 and you want to make the value 6. You would enter the following command and the current value of the variable would be returned:

```
$ gv_write users 6
5
```

Then you would enter the `gv_read` command to see the changed value:

```
$ gv_read users
6
```

If the variable type is string and the value specifies to contain spaces, you should enclose the value in quotes. For example:

```
$ gv_write solution "world peace"
```

SEE ALSO Gv_write

FUNCTION Gv_write, Gv_writev

SYNTAX

```
int Gv_write(name, value)
char *name;
int value;

int Gv_writev(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name`	The name of the global variable
`value`	The new value assigned to the variable
`oldvalue`	The pointer location into which the original value should be stored

Comments

The `Gv_write()` and `Gv_writev()` functions assign a new value to the specified variable. `Gv_write()` is used if the variable is an integer, and `Gv_writev()` is used if the variable is not an integer. The `value` parameter specifies the new value. In the `Gv_writev()` function, the `oldvalue` parameter specifies the pointer location into which the original value should be stored.

You can enter these functions into your script when editing the script file.

RETURN VALUE The `Gv_write()` function returns the original value of the specified variable and the `Gv_writev()` function copies the original value to a pointer location. After the new value has been assigned to the variable, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

To assign a new value to the variable with the name `count`, use the following example if `count` is an integer:

```
Gv_alloc("count", "int");
Gv_write("count", 12);
```

To assign a new value to the non-integer variable `balance`, use:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_writev("balance", 120.75, &curbalance);
```

SEE ALSO

gv_write

COMMAND gv_xor

SYNTAX `gv_xor name value`

DESCRIPTION The `gv_xor` command is entered at the shell prompt and updates the value of a specified variable by applying a bit-wise EXCLUSIVE-OR masking to a variable. The parameter `value` is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_xor

FUNCTION **Gv_xor, Gv_xorv**

SYNTAX

```
int Gv_xor(name, value)
char *name;
int value;

int Gv_xorv(name, value, oldvalue)
char *name;
```

DESCRIPTION

Parameters

`name` The name of the global variable

`value` The operand for the operation

`oldvalue` The pointer location where the original value should be stored

Comments

The `Gv_xor()` and `Gv_xorv()` functions update the value of a specified variable by applying a bit-wise EXCLUSIVE-OR masking to a variable. You can insert these functions into your script when editing your script file.

`Gv_xor()` is used if the variable is an integer, and `Gv_xorv()` is used if the variable is not an integer. The parameter `value` is the operand for the operation, and the parameter `oldvalue` (for `Gv_xorv()`) specifies the pointer location where the original value should be stored.

RETURN VALUE `Gv_xor()` returns the original value of the specified variable and `Gv_xorv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO gv_xor

FUNCTION **Hostname**

SYNTAX

```
Hostname(lognum, hostname)
int lognum;
char *hostname;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`hostname` The name of the host machine that includes the database server

Comments

This function is inserted into the script to inform the Logon() connection on which machine the database(s) and server reside. It will occur in the script before a `Logon()` function.

During script execution, the `Hostname()` function is used in the process to access the SUT by specifying the name of the host machine where the database server being tested resides.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLE The following example demonstrates this function used in a script:

```
Hostname(LOG1, "SUT");  /* the DBserver resides on machine name SUT */
```

SEE ALSO AppName, Language, Password, Servername, Username

0095796 "10260

FUNCTION InitialWindow

SYNTAX

```
InitialWindow(task, window, x, y, width, height)
int task;
char *window;
int x,y,width,height;
```

DESCRIPTION

Parameters

`task`	The order of the open program as a task
`window`	The title of the open window
`x,y`	The on screen x,y coordinates of the top, left corner of the window
`width` `height`	The width and height of the window on screen

Comments

This function is an EMPOWER/CS function that is inserted into the script during Capture after the `Beginscenario()` function. `InitialWindow()` records the state of your Windows desktop by listing all active program windows at the time the Capture session began. Therefore, multiple `InitialWindow()` functions can be recorded into the script and are listed consecutively.

The first parameter, `task`, indicates the order of the specified program in the MS Windows task list. The position in the task list is important for users who capture using "**Fast-Tab**" task switching. A `task` parameter of zero is a reserved value that records the screen resolution during Capture and verifies the resolution during script execution in Display mode.

The `window` parameter identifies the title of the window. The `x,y` parameters indicate the top left x,y coordinates of the window's position on screen. The `width` and `height` parameters determine the size of the window as displayed. If `width` and `height` are both zero, the window is

minimized. If the `x,y` parameters are both zero and `width` is `MAXWIDTH` and `height` is `MAXHEIGHT`, the window is maximized. If neither of these conditions apply, the window is in a normal state.

During script execution in Display mode, `InitialWindow()` attempts to locate the Windows listed as its parameters and place them in the same positions as captured. The function does not apply to Non-Display mode script execution.

If the specified programs are not open during script execution, the log file will record an error but script execution will continue. Therefore, before you execute your script in Display mode, your desktop should be in the same state with the same applications open as when you captured the script.

Note: You should not attempt to remove or edit this function in your script file because you change the state of the windows desktop as it was captured, and, therefore run the risk of breaking the script.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates `InitialWindow()` in a script file:

```
InitialWindow(0,"Desktop",0,0,640,480);          /* Screen resolution */
InitialWindow(1,"MS-DOS Prompt",21,408,0,0);     /* Minimized window */
InitialWindow(2,"File Manager",0,0,MAXWIDTH,MAXHEIGHT);/* Maximized window */
InitialWindow(3,"Program Manager",36,26,545,333); /* Normal window */
```

The following example demonstrates the log file of an executed script that encountered an error because the desktop was not restored to its captured state:

```
>>>     InitialWindow(2,"File Manager",0,0,MAXWIDTH,MAXHEIGHT)
        Warning: Unable to find window
```

SEE ALSO CurrentWindow

00969794.102600

FUNCTION **Inote**

SYNTAX

```
Inote(n)
int n;
```

DESCRIPTION

Parameters

`n` An integer to be displayed in Monitor

Comments

The `Inote()` function specifies a message that will appear in the "Note" column of Monitor View 5. The `n` parameter is an integer that will be displayed. When a script is executed with the `Inote()` function, an `Inote()` statement is placed into the script's log file.

You must insert the `Inote()` function into the script when editing your script file.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates using `Inote()` in a script to identify the number of a loop as it executes:

```
for (i=1; i≤10; i++){
   Inote(i); /* identify loop number */
...
}
```

SEE ALSO

Note

FUNCTION **KeyDown**

SYNTAX

```
void KeyDown(key)
unsigned int key;
```

DESCRIPTION

Parameters

`key` A Microsoft Windows virtual key code value

Comments

The `KeyDown()` function is inserted in the script file automatically during Capture when the user presses the specified key. During script execution in Display mode, this function is used to emulate pressing the specified key down. In Non-Display mode, this function is used to emulate the type delay for pressing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some examples of valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, `VK_UP`, and `VK_ESCAPE`.

The `KeyDown()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you edit a `KeyDown()` function, you also must edit the associated `KeyUp()` function.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various key events where the user pressed the left arrow key, the shift key, the tab key, control key, and escape key:

```
WindowRcv("Pt");
KeyPress(VK_LEFT);
KeyDown(VK_SHIFT);
KeyPress(VK_TAB);
KeyUp(VK_SHIFT);
KeyDown(VK_CONTROL);

Type("^I^I");

KeyUp(VK_CONTROL);

KeyPress(VK_ESCAPE);
```

SEE ALSO KeyPress, KeyUp, SysKeyDown, SysKeyPress, SysKeyUp

FUNCTION **KeyPress**

SYNTAX

```
void KeyPress(key)
unsigned int key;
```

DESCRIPTION

Parameters

`key` A Microsoft Windows virtual key code value

Comments

A `KeyPress()` function is translated into a down/up key sequence. A `KeyPress()` function is inserted in the script file automatically during Capture when no other user events occur between a KeyDown/Up pair. If the key is the same for consecutive KeyDown and KeyUp events, the key events will automatically be translated into a KeyPress.

During script execution in Display mode, this function is used to emulate the pressing and releasing of the specified key. In Non-Display mode, this function is used to emulate the type delay for pressing and releasing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

Key events may be edited within the script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various key events where the user pressed the left arrow key, the shift key, the tab key, control key, and escape key:

```
WindowRcv("Pt");
KeyPress(VK_LEFT);
KeyDown(VK_SHIFT);
KeyPress(VK_TAB);
KeyUp(VK_SHIFT);
KeyDown(VK_CONTROL);

Type("^I^I^I");

KeyUp(VK_CONTROL);

KeyPress(VK_ESCAPE);
```

SEE ALSO KeyDown, KeyUp, SysKeyDown, SysKeyPress, SysKeyUp

FUNCTION **KeyUp**

SYNTAX

```
void KeyUp(key)
unsigned int key;
```

DESCRIPTION

Parameters

`key` A Microsoft Windows virtual key code value

Comments

The `KeyUp()` function is inserted in the script file automatically during Capture and indicates that a non-printable keyboard key on the PC was released. During script execution in Display mode, this function is used to emulate releasing the specified key. In Non-Display mode, this function is used to emulate the type delay for releasing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

This function may be edited within your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you edit the `KeyUp()` function, you also must edit the associated `KeyDown()` function.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various key events where the user pressed the left arrow key, the shift key, the tab key, control key, and escape key:

```
WindowRcv("Pt");
KeyPress(VK_LEFT);
KeyDown(VK_SHIFT);
KeyPress(VK_TAB);
KeyUp(VK_SHIFT);
KeyDown(VK_CONTROL);

Type("^I^I^I");

KeyUp(VK_CONTROL);

KeyPress(VK_ESCAPE);
```

SEE ALSO KeyDown, KeyPress, SysKeyDown, SysKeyPress, SysKeyUp

FUNCTION **LeftButtonDown**

SYNTAX

```
void LeftButtonDown(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `LeftButtonDown()` function is inserted in the script file automatically during Capture when the user depresses the left mouse button. During script execution in Display mode, this function is used to emulate pressing the left mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed during Capture.

The `LeftButtonDown()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various left button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);
```

(continued on following page . . .)

```
AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

LeftButtonPress(338,220);
```

Sometimes, a comment is inserted by EMPOWER/CS before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
LeftDblPress(276,345);
```

SEE ALSO LeftButtonPress, LeftButtonUp, LeftDblPress

FUNCTION LeftButtonPress

SYNTAX

```
void LeftButtonPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

`x` The on screen x coordinate of the PC mouse

`y` The on screen y coordinate of the PC mouse

Comments

The `LeftButtonPress()` function indicates when the left button on the PC mouse was pressed down and released during Capture. A `LeftButtonPress()` function is inserted in the script when no other user events occur between a LeftButtonDown/Up pair. If the x,y coordinates are the same for consecutive LeftButtonDown and LeftButtonUp events, the left mouse button events will be translated into a `LeftButtonPress()`.

During script execution in Display mode, this function is used to emulate pressing and releasing the left mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed.

The `LeftButtonPress()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various left button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

LeftButtonPress(338,220);
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
LeftDblPress(276,345);
```

SEE ALSO LeftButtonDown, LeftButtonUp, LeftDblPress

FUNCTION **LeftButtonUp**

SYNTAX

```
void LeftButtonUp(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `LeftButtonUp()` function is inserted in the script file automatically during Capture when the user releases the left mouse button. During script execution in Display mode, this function is used to emulate releasing the left mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was released during Capture.

The `LeftButtonUp()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various left button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

LeftButtonPress(338,220);
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
LeftDblPress(276,345);
```

SEE ALSO LeftButtonDown, LeftButtonPress, LeftDblPress

FUNCTION LeftDblPress

SYNTAX

```
void LeftDblPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `LeftDblPress()` function is inserted in the script file automatically during Capture to indicate that a double press of the left mouse button occurred. During script execution in Display mode, this function is used to emulate a double press of the left mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was double-clicked during Capture.

This function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftDblPress(213,111);


AppWait(0.55);
WindowRcv("Pt");
```

```
LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonUp(333,219);
LeftButtonPress(338,221);

AppWait(0.17);
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
LeftDblPress(276,345);
```

SEE ALSO LeftButtonDown, LeftButtonPress, LeftButtonUp

FUNCTION **Log**

SYNTAX

```
void Log(str)
char *str;
```

DESCRIPTION

Parameters

`str` A null-terminated string recorded into the log file of an executed script

Comments

You can insert the `Log()` function into your script when editing the script file. This function allows you to place additional information in a log, such as the value of a variable.

`Log()` records the parameter `str` in the log file. The parameter `str` is a null-terminated string and is recorded in the log file in the following format.

```
>>>  16 Log("str")
```

Note: The number following the ">>>" is the line number corresponding to a line in the script file.

RETURN VALUE (not applicable)

SEE ALSO Inote, Note

FUNCTION **Logoff**

SYNTAX

```
Logoff(lognum)
int lognum;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

Comments

The `Logoff()` function is inserted into the script file when a logon structure is closed. A logoff usually occurs when a user terminates a connection with the SUT.

During script execution, this function closes the logon structure that was opened with the corresponding `Logon()`. All active cursors on the specified logon structure must be closed before the logon structure will close..

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates a `Logoff()` function in a script:

```
Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE1);

Endscenario("script1");
```

SEE ALSO Close, Closeenv, Logon

FUNCTION **Logon**

SYNTAX

```
Logon(dbnum, lognum)
int dbnum, lognum;
```

DESCRIPTION

Parameters

`dbnum` The number of the database environment

`lognum` An identifier of a logon communication structure

Comments

The `Logon()` function is inserted into the script when a logon connection to the database is opened. During script execution, this function opens a communication link to the database for a particular database environment.

The database environment for the specified `dbnum` must have been opened with `Openenv()` before a `Logon()` call will execute. The `Logon()` function provides access to the database so that a cursor structure can be opened to process the SQL statement. Therefore, `Logon()` will occur after `Openenv()` and before `Open()` in the script. The specifications for `Username()` and `Password()` also must be called in the script before a `Logon()` function will successfully execute.

Opening more than one logon connection from a database environment is possible. The first logon structure opened will be numbered by EMPOWER/CS as LOG1, the second as LOG2, etc.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following script segment demonstrates the process of logging on to the database, ORACLE, from the logon structure LOG1:

```
Type("scott^Itiger^ITRAMP^M")

Username(LOG1, "scott@TRAMP");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);

AppWait(0.05);

Think(2.35);

...


Open(LOG1,CUR1);

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
```

SEE ALSO

Logoff, Open, Openenv, Password, Username

FUNCTION **MiddleButtonDown**

SYNTAX

```
void MiddleButtonDown(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `MiddleButtonDown()` function is inserted in the script file automatically during Capture when the user depresses the middle mouse button. During script execution in Display mode, this function is used to emulate pressing the middle mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed.

The `MiddleButtonDown()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you change the `MiddleButtonDown()` function, you also must change the associated `MiddleButtonUp()` function.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);

AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

MiddleButtonDown(340,216);
MiddleButtonUp(333,219);
MiddleButtonPress(338,221);

AppWait(0.17);
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
MiddleDblPress(276,345);
```

SEE ALSO MiddleButtonPress, MiddleButtonUp, MiddleDblPress

FUNCTION **MiddleButtonPress**

SYNTAX

```
void MiddleButtonPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `MiddleButtonPress()` function indicates when the middle button on the PC mouse was pressed down and released during Capture. A `MiddleButtonPress()` is inserted in the script when two consecutive MiddleButtonDown/Up events do not have any other user events between them.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed during Capture.

During script execution in Display mode, this function is used to emulate pressing and releasing the middle mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `MiddleButtonPress()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

MiddleButtonDown(340,216);
MiddleButtonUp(333,219);
MiddleButtonPress(338,221);

AppWait(0.17);
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
MiddleDblPress(276,345);
```

SEE ALSO MiddleButtonDown, MiddleButtonUp, MiddleDblPress

FUNCTION MiddleButtonUp

SYNTAX

```
void MiddleButtonUp(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `MiddleButtonUp()` function is inserted in the script file automatically during Capture when the user releases the middle mouse button. During script execution in Display mode, this function is used to emulate releasing the middle mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was released during Capture.

The `MiddleButtonUp()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you edit this function, you also must change the associated `MiddleButtonDown()` function.

RETURN VALUE (not applicable)

EXAMPLES

The following example demonstrates various button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

MiddleButtonDown(340,216);
MiddleButtonUp(333,219);
MiddleButtonPress(338,221);

AppWait(0.17);
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
MiddleDblPress(276,345);
```

SEE ALSO

MiddleButtonDown, MiddleButtonPress, MiddleDblPress

FUNCTION **MiddleDblPress**

SYNTAX

```
void MiddleDblPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

x The on screen x coordinate of the PC mouse

y The on screen y coordinate of the PC mouse

Comments

The `MiddleDblPress()` function is inserted in the script file automatically during Capture to indicate that a double press of the middle mouse button occurred. During script execution in Display mode, this function is used to emulate a double press of the middle mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was double-clicked during Capture.

The `MiddleDblPress()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES

The following example demonstrates various button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

MiddleButtonDown(340,216);
MiddleButtonPress(333,219);
MiddleButtonUp(338,221);

AppWait(0.17);
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
MiddleDblPress(276,345);
```

SEE ALSO

MiddleButtonDown, MiddleButtonPress, MiddleButtonUp

FUNCTION **Note**

SYNTAX

```
Note(str)
unsigned char *str;
```

DESCRIPTION

Parameters

`str` The text of a message to be displayed in Monitor

Comments

You must insert the `Note()` function into the script when editing. The `Note()` function specifies a message that will appear in the "Note" column of Monitor View 5. The `str` parameter is the text of the message and this message must be contained within double quotes. When a script executes, all `Note()` statement will be listed in the log file.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Inote, Log

FUNCTION **Open**

SYNTAX

```
Open(lognum, curnum)
int lognum, curnum;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`curnum` An identifier of a cursor communication structure

Comments

The `Open()` function is inserted into the script file when a cursor communication structure is opened. It is inserted after a `Logon()` call and before database processing begins.

During script execution, this function enables the script to interact with the database over a communication structure called a cursor. A cursor structure is opened so that the SQL statement can be processed. The `Open()` function opens a cursor from the logon connection with the database and multiple cursors can be opened for a particular logon structure.

The parameter `lognum` specifies the logon structure from which the cursor was opened. The parameter `curnum` specifies the cursor on which the script will be operating.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example script segment demonstrates opening a cursor, CUR1, from the logon structure LOG1:

```
Logon(ORACLE1, LOG1);

AppWait(0.05);

Think(2.35);

...

Open(LOG1,CUR1);

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
Define(CUR1, "1", STRING, 5);
Define(CUR1, "2", STRING, 11);
Define(CUR1, "3", STRING, 10);
Define(CUR1, "4", STRING, 5);
Define(CUR1, "5", STRING, 10);
Define(CUR1, "6", STRING, 9);
Define(CUR1, "7", STRING, 9);
Define(CUR1, "8", STRING, 3);
Exec(CUR1);

Fetch(CUR1);
GetNextRow(CUR1);
```

SEE ALSO

Close, Logon, Openenv

FUNCTION **Openenv**

SYNTAX

```
Openenv(dbnum, v)
int dbnum, v;
```

DESCRIPTION

Parameters

`dbnum`	The number of the database environment
`v`	The version number of the database

Comments

The `Openenv()` function is inserted into the script when a database environment is opened. During script execution, `Openenv()` opens an environment specifying the database and database version to be used for the emulation.

Log on connections to the database can be made only from an open environment.

In some cases, multiple databases may be accessed and activity for each database may occur concurrently. Therefore, more than one `Openenv()` may be captured into a script depending on the captured application activity.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates how `Openenv()` may appear in a script file. In this example, a Version V6V7 Oracle database was specified:

```
Openenv(ORACLE1, VERSIONV6V7);
```

SEE ALSO

Closenv, Logon, Open

FUNCTION **Paceconstant**

SYNTAX

```
Paceconstant(str, n)
char *str;
int n;
```

DESCRIPTION

Parameters

`str` An argument naming the pace and defining the speed of the pace

`n` The number of seconds the script should delay since the last call to `Paceconstant()`

Comments

Pacing functions can be inserted into your script file to control the pace of the script. These functions cause the script to pause long enough to make the script send transactions to the SUT at a predetermined throughput. Typically used in a loop, these functions may be nested to permit controlled throughput of transactions within a larger transaction.

`Paceconstant()` causes transactions to be submitted at a constant throughput. It accepts an argument naming the pace and defining the speed of the pace. The second argument to `Paceconstant()` defines the number of seconds that the script should take since the last call to `Paceconstant()`. The first call to a pacing function does not delay; it is used as a starting point for the subsequent calls. For this reason, the first call to a pacing function is often made with arguments of 0 to define the speed.

Note: You are permitted to nest pacing functions in a script. If you do so, we recommend setting the starting point for a pace explicitly, i.e. with the speed argument of 0. This will ensure that an inner loop executed at a fixed pace will begin execution immediately every time that the loop begins.

RETURN VALUE This function returns the amount of time delayed.

SEE ALSO Pacetne, Paceuniform

FUNCTION **Pacetne**

SYNTAX

```
Pacetne(str, min, ave, max)
char *str;
int min, ave, max;
```

DESCRIPTION

Parameters

`str`	An argument naming the pace and defining the speed of the pace
`min`	An integer specifying minimum delay for the speed of the pace
`ave`	An integer specifying the average delay for the speed of the pace
`max`	An integer specifying maximum delay for the speed of the pace

Comments

Pacing functions can be inserted into your script file to control the pace of the script. These functions cause the script to pause long enough to make the script send transactions to the SUT at a predetermined throughput. Typically used in a loop, these functions may be nested to permit controlled throughput of transactions within a larger transaction.

`Pacetne()` has an average throughput that will be maintained, but submission of transactions occur at frequencies other than that defined by a constant distribution.

`Pacetne()` accepts three arguments to identify the speed of the pace - a minimum, average, and maximum delay. The average will be maintained during sustained execution of the script. Each time `Pacetne()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a truncated, negative exponential distribution defined by the three values. In a typical such distribution, the average is relatively close to the minimum. For

example, if `Pacetne("query", 7.0, 10.0, 20.0)` is used in a script, a sequence of 7, 8, 10, and 15 second delays is possible. The average of the delays is still ten seconds. `Pacetne()` will select the values for delay accurate to 1/10th of a second, so 9.2 seconds is a possible value.

Note: You are permitted to nest pacing functions in a script. If you do so, we recommend setting the starting point for a pace explicitly, i.e. with the speed argument of 0. This will ensure that an inner loop executed at a fixed pace will begin execution immediately every time that the loop begins.

RETURN VALUE This function returns the amount of time delayed.

SEE ALSO Paceconstant, Paceuniform

FUNCTION **Paceuniform**

SYNTAX

```
Paceuniform(str, min, max)
char *str;
int min, max;
```

DESCRIPTION

Parameters

`str`	An argument naming the pace
`min`	The minimum delay for the speed of the pace
`max`	The maximum delay for the speed of the pace

Comments

`Paceuniform()` has an average throughput that will be maintained, but submission of transactions occurs at frequencies other than that defined by a constant distribution.

`Paceuniform()` accepts two arguments to identify the speed of the pace - a minimum delay and a maximum delay. The average of the two will be maintained during sustained execution of the script. Each time `Paceuniform()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a uniform distribution between the minimum and maximum values. For example, if `Paceuniform("query", 8.0, 12.0)` is used in a script, a sequence of 9, 11, 8, 10, and 12 second delays is possible (assuming the `query` has a response time of zero seconds.) The average of the delays is still ten seconds. `Paceuniform()` will select the values for delay accurate to 1/100th of a second, so 9.27 seconds is a possible value.

RETURN VALUE This function returns the amount of time delayed.

SEE ALSO Paceconstant, Pacetne

FUNCTION **Parse**

SYNTAX

```
Parse(curnum, sqlstmt, .../* args */)
int curnum;
char *sqlstmt, *args;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure opened with the associated `Open()`
`sqlstmt`	The SQL statement to be parsed
`args`	Optional variable arguments

Comments

This function is inserted into the script when a SQL statement is parsed and sent to the SUT. The `Parse()` function parses a SQL statement and associates it with a cursor identified in the parameter `curnum`. The parameter `sqlstmt` is a string that lists the SQL statement.

During script execution, `Parse()` sends the SQL statement to the SUT and verifies that the syntax of the SQL statement is compatible with the SUT. The SQL statement is stored on the SUT to be executed with a subsequent `Exec()` call. This operation prepares the SUT for processing of the SQL statement by such functions as `Define()` or `Bind()`, `Describe()`, `Data()`, etc.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `Parse()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments must be string pointers. The string conversion specification is provided by the characters `%s`.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates selecting the data for a query in a SQL statement for the cursor structure CUR1:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
```

SEE ALSO Exec, Fetch

FUNCTION **Password**

SYNTAX

```
Password(lognum, password)
int lognum;
char *password;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`password` The user password used to access the database

Comments

To successfully access or logon to the database, a Username and Password must be entered. The `Password()` function is inserted into the script file when a user password is entered to logon to the database. During script execution, this function is used in the process of accessing the SUT. This function will occur in the script before a `Logon()` function.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES In the following example, the script will attempt to logon to the database `ORACLE` on the logon structure `LOG1` with the password `tiger`:

```
Username(LOG1, "scott@FOO");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);
```

SEE ALSO AppName, Hostname, Language, Servername, Username

FUNCTION **Pause**

SYNTAX

```
Pause(lognum, str)
int lognum, str;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`str` The id of the transaction to be paused

Comments

The `Pause()` function specifies an application-defined database transaction in progress that is to be suspended until a `Continue()` call is executed. This function is captured into the script when the client application instructs the database to halt execution of the current transaction. It will be inserted following the associated `Transaction()` function.

The transaction id, `str` parameter, assigned in the `Pause()` function is lated used in the `Continue()` function to continue work on the transaction.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates the Pause() function within a script:

```
BeginTransaction(LOG1);
....
Pause(LOG1, TRANS2);

Continue(LOG1, TRANS1);
Commit(TRANS1);
```

0965794E.102600

SEE ALSO BeginTransaction, Continue

FUNCTION Pointerrate

SYNTAX

```
double Pointerrate(n)
double n;
```

DESCRIPTION

Parameters

n The mouse pointer speed measured in points per second

Comments

`Pointerrate()` sets the emulated mouse pointer speed for a script. Pointer speed is measured in points per second which is specified in the parameter n. The default pointer rate `Pointerrate(100)` which is listed at the top of each script created by EMPOWER/CS. You may change this default when editing the script file. Multiple `Pointerrate()` functions may be used in a script to specify, for example, different pointer rates when using different applications.

During script execution, each time that the emulated user is supposed to be moving the PC mouse, the script will pause a length of time defined as the number of points to be moved times the specified pointer rate. For example, if the emulated user must move the mouse 100 points and the pointer rate is specified as 50 points per second (`Pointerrate(50)`) then the script will pause two seconds during each mouse point from one location to the next specified location.

Care must be taken when inserting `Pointerrate()` functions into a script. The response time statistics for scenarios and functions are generated from the user's perspective. Therefore, they will take into account the time it takes for the emulated user to move the PC mouse. This is particularly important when you are comparing the performance of similar scripts. Any `Pointerrate()` functions inserted into one script must be inserted in the same locations in the other script(s) to provide a meaningful comparison.

error if you set rate at less than zero, script will abort and you will receive an error message.

RETURN VALUE This function returns the old pointer rate value.

EXAMPLES In the following example, the `Pointerrate()` is set at the beginning of the script at 150 points per second:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Typerate(5);              /* Typing delay in CPS */
Pointerrate(150);         /* Pointerrate in PPS */
Thinkuniform(1,2.5);      /* Think delay */
Seed(getpid());           /* Seed random number generator */
Timeout(300, CONTINUE);   /* What to do if function takes too long */
Dberror(CONTINUE);        /* What to do on Database errors */
Unset(NOTIFY);            /* Don't display warnings. I'll use Mon to find them */
```

SEE ALSO Typerate

FUNCTION **Range**

SYNTAX

```
int Range(min, max)
int min, max;
```

DESCRIPTION

Parameters

`min` An integer specifying the minimum value of the range

`max` An integer specifying the maximum value of the range

Comments

`Range()` often is used with the `Sleep()` function to perform random execution delays. This function is inserted into the script file during script editing.

`Range()` returns a random number.

RETURN VALUE `Range()` returns a random integer number between or equal to `min` and `max`.

EXAMPLES The following script segment is an example of how to perform a random execution delay from 0 to 60 seconds.

```
Sleep(Range(0, 60));
```

SEE ALSO Seed, Sleep

FUNCTION RightButtonDown

SYNTAX

```
void RightButtonDown(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

`x` The on screen x coordinate of the PC mouse

`y` The on screen y coordinate of the PC mouse

Comments

The `RightButtonDown()` function is inserted in the script file automatically during Capture when the user depresses the right mouse button down. During script execution in Display mode, this function is used to emulate pressing the right mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed.

The `RightButtonDown()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you change this function, you must be sure to change the associated `RightButtonUp()` function.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various mouse button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

RightButtonDown(340,216);
RightButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
RightDblPress(276,345);
```

SEE ALSO RightButtonPress, RightButtonUp, RightDblPress

FUNCTION RightButtonPress

SYNTAX

```
void RightButtonPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

`x` The on screen x coordinate of the PC mouse

`y` The on screen y coordinate of the PC mouse

Comments

The `RightButtonPress()` function is inserted in the script file automatically during Capture when the right button on the PC mouse was pressed down and released. A `RightButtonPress()` is inserted in the script when two consecutive RightButtonDown/Up events do not have any other user events between them.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was pressed.

During script execution in Display mode, this function is used to emulate pressing the left mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `RightButtonPress()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various mouse button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

RightButtonDown(340,216);
RightButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
RightDblPress(276,345);
```

SEE ALSO RightButtonDown, RightButtonUp, RightDblPress

FUNCTION **RightButtonUp**

SYNTAX
```
void RightButtonUp(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

`x` The on screen x coordinate of the PC mouse

`y` The on screen y coordinate of the PC mouse

Comments

The `RightButtonUp()` function is inserted in the script file automatically during Capture when the user releases the right mouse button. During script execution in Display mode, this function is used to emulate pressing the right mouse button down. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was released.

The `RightButtonUp()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script. If you change this function, you also must change the associated `RightButtonDown()` function.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various mouse button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

RightButtonDown(340,216);
RightButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
RightDblPress(276,345);
```

SEE ALSO RightButtonDown, RightButtonPress, RightDblPress

FUNCTION **RightDblPress**

SYNTAX

```
void RightDblPress(x,y)
unsigned int x,y;
```

DESCRIPTION

Parameters

`x` The on screen x coordinate of the PC mouse

`y` The on screen y coordinate of the PC mouse

Comments

The `RightDblPress()` function is inserted in the script file automatically during Capture to indicate that a double press of the right mouse button occurred. During script execution in Display mode, this function is used to emulate a double click of the right mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the `x,y` parameters.

The `x,y` parameters indicate the position of the mouse on screen at the time the button was double-clicked during Capture.

The `RightDblPress()` function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various mouse button activities:

```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);


AppWait(0.55);
WindowRcv("Pt");

LeftButtonUp(222,195);


AppWait(0.38);
WindowRcv("Pt");

RightButtonDown(340,216);
RightButtonUp(333,219);

AppWait(0.17);
WindowRcv("Pt");

RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */
RightDblPress(276,345);
```

SEE ALSO RightButtonDown, RightButtonPress, RightButtonUp

FUNCTION **Rollback**

SYNTAX

```
Rollback(num)
int num;
```

DESCRIPTION

Parameters

num A logon or cursor communication structure

Comments

The `Rollback()` function is inserted into the script when database transactions are rolled back from the database since data was last committed or an `Open()` occurred. `Rollback()` restores the database to its original state since the last `Commit()` was executed.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

This example demonstrates a `Rollback()` function in a script file:

```
Commit(CUR1);

/* insert data into database */
Parse(CUR1, "insert empno, ename, empjob into emp_table");

Bind(CUR1, "empno", INT, 4);
Bind(CUR1, "ename", STRING, 30);
Bind(CUR1, "empjob", STRING, 20);

/* 123 refers to empno, Smith -- ename, typist -- empjob */
Data(CUR1, "123|Smith|typist");

Exec(CUR1);

Rollback(CUR1);
```

SEE ALSO

Commit

FUNCTION **Seed**

SYNTAX

```
int Seed(n)
int n;
```

DESCRIPTION

Parameters

n The value used to seed the random number generator

Comments

`Seed()` seeds the random number generator used by EMPOWER/CS that produces random think time values.

If you wish to generate the same sequence of think times during repeated script executions, you should always seed the random number generator in the script with the same value. If you wish to generate different sequences of think times for each script in a multi-user emulation, you should seed each random number generator in each script with a different value.

RETURN VALUE Seed returns a value of n.

EXAMPLES The following example `Seed()` function seeds the random number generator from an argument specified on the command line used to execute the script:

```
Seed(atoi(argv[3]));
```

SEE ALSO Thinkconstant, Thinktne, Thinkuniform, Range

FUNCTION	**Servername**

SYNTAX

```
Servername(lognum, servername)
int lognum;
char *servername;
```

DESCRIPTION

Parameters

`lognum`	An identifier of a logon communication structure
`servername`	The name of the database server

Comments

This function is inserted into the script when a server name was specified for a logon connection. It will occur in the script before a `Logon()` call.

`Servername()` sets the name of the database server to which the user connected.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

The following example demonstrates the `Servername()` function in a script:

```
Servername(LOG1, "DBSERVER");
Username(LOG1, "joe");
Password(LOG1, "xapq");
Logon(SYBASE, LOG1);
```

SEE ALSO

AppName, Hostname, Language, Password, Username,

FUNCTION **Set**

SYNTAX

```
long Set(n)
long n;
```

DESCRIPTION

Parameters

n The EMPOWER/CS option turned on

Comments

`Set()` turns on EMPOWER/CS options during script execution. Valid options are listed below:

OPTION	DESCRIPTION
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in a `WindowRcv()`
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	enable logging

Default options are `LCMD` and `LOGGING`.

RETURN VALUE `Set()` returns a long value that contains previously set options.

EXAMPLES The following script segment causes flushing of the remaining buffer after the pattern is found and immediate recording of every response character in the log file.

```
Set(FLUSH);/* turn flush buffer on*/
Set(NOBUF);/* turn immediate recording on */
```

SEE ALSO Unset

FUNCTION **SetIntVar**

SYNTAX

```
SetIntVar(curnum, var, value)
int curnum;
char *var;
int value;
```

DESCRIPTION

Parameters

`curnum` A cursor communication structure

`var` The name of the variable

`value` The new value to be assigned to the variable

Comments

The `SetIntVar()` function assigns a new value to the specified integer variable in the parameter var. The new value is assigned in the parameter value. This function is inserted into your script file when editing and should occur before `Exec()` or after `GetVar()` or `GetNextRow()`.

RETURN VALUE After the new value has been assigned to the specified variable, the original value of the variable is returned.

SEE ALSO GetIntVar, GetVar, SetVar

FUNCTION **SetVar**

SYNTAX

```
SetVar(curnum, var, value)
int curnum;
char *var, *value;
```

DESCRIPTION

Parameters

`curnum` A cursor communication structure

`var` The name of the variable

`value` The new value to be assigned to the variable

Comments

The `SetVar()` function assigns a new value to the specified variable in the parameter var. The new value is assigned in the parameter value. You can insert this function into your script file when editing.

RETURN VALUE After the new value has been assigned to the specified variable, the original value of the variable is returned.

EXAMPLES The following example demonstrates using this function in a script:

```
char *empname, *empno;

...

Parse(CUR1, "insert ename, empno, empjob into
employee_table");

Bind(CUR1, "ename", STRING, 50);
Bind(CUR1, "empno", INT, 4);
Bind(CUR1, "empjob", STRING, 30);
```

(continued on following page . . .)

0057594.1020000

```
Data(CUR1, "Smith|234|typist");

/* inserts "Smith|234|typist" into database */
Exec(CUR1);

empno=GetVar(CUR1, "empno");

for (i=0;i<5;i++){
  /* this increments the value of empno by 1 everytime */
  *empno=*(int *)empno+i;
  SetVar(CUR1, "empno", empno);

  /* this reads in a unique name from a name file everytime */
  empname=Fioreadfield("namefile");
  SetVar(CUR1, "empname", empname);

  /* inserts "new name|empno+1|typist" into database */
  Exec(CUR);
}
```

SEE ALSO GetIntVar, GetVar, SetIntVar

FUNCTION **Sleep**

SYNTAX
```
double Sleep(n)
double n;
```

DESCRIPTION *Parameters*

n The number of seconds to suspend script execution

Comments

`Sleep()` suspends script execution for n seconds simulating a think delay.

RETURN VALUE `Sleep()` returns n.

EXAMPLES In the following example, `Sleep()` specifies that script execution will suspend for 30 seconds:

```
Sleep(30);
```

You can also specify a range for the `Sleep()` parameter:

```
Sleep(Range(40,60));
```

SEE ALSO Think, Thinkconstant, Thinktne, Thinkuniform

FUNCTION **Sql**

SYNTAX

```
Sql(curnum, sqlstmt, .../* args */)
int curnum;
char *sqlstmt, *args;
```

DESCRIPTION

Parameters

`curnum`	An identifier of the cursor structure opened with the associated `Open()`
`sqlstmt`	The SQL statement to be parsed
`args`	Optional variable arguments

Comments

This function is inserted into the script when a SQL statement is parsed and sent to the SUT. The `Sql()` function parses a SQL statement and associates it with a cursor identified in the parameter `curnum`. The parameter `sqlstmt` is a string that lists the SQL statement.

During script execution, `Sql()` sends the SQL statement to the SUT and verifies that the syntax of the SQL statement is compatible with the SUT. The SQL statement is stored on the SUT to be executed with a subsequent `Exec()` call. This operation prepares the SUT for subsequent processing of the SQL statement with such functions as `Define()` or `Bind()`, `Describe()`, `Data()`, etc.

The `Sql()` statement is called before an `Exec()` in the script.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `Sql()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments

must be string pointers. The string conversion specification is provided by the characters %s.

RETURN VALUE — If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES — The following example demonstrates selecting the data for a query in a SQL statement for the cursor structure CUR1:

```
Sql(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
```

SEE ALSO — Exec, Fetch, Parse

FUNCTION **SqlExec**

SYNTAX

```
SqlExec(lognum, sqlstmt, .../* args */)
int lognum;
char *sqlstmt, *args;
```

DESCRIPTION

Parameters

`lognum` Specifies a logon communication structure

`sqlstmt` Lists the SQL statement to be executed

`args` Optional variable arguments

Comments

A `SqlExec()` function is inserted into your script when a SQL statement was executed that contained no input or output variables. The `SqlExec()` function executes the SQL statement on the SUT completing four database operations. It opens a cursor, parses a SQL statement, executes the `Parse()`, and closes the cursor.

When this function is used, no operations such as `Describe()`, `Bind()`, `Define()`, `Fetch()`, etc., are completed. For example, it may be used when a SQL statement requires only the joining of two tables.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `SqlExec()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments must be string pointers. The string conversion specification is provided by the characters `%s`.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO Exec, Parse, Sql

FUNCTION **Suspend**

SYNTAX `Suspend();`

DESCRIPTION

`Suspend()` is used during script execution to suspend a script until the signal `SIGRESUME` arrives. Suspending script execution is useful when multiple scripts are executed simultaneously. Use of the suspend feature permits control and synchronization of concurrent scripts.

If the `LCMD` (a default option) option is set with `Set()`, a suspended message with the time stamp of the suspension will be recorded in the log file. A sample of the suspend message in the log file is shown below:

```
>>>     33 Suspend() 13:37:06.96
```

Once a script has been suspended, execution can be resumed in three ways. First, if the script was started with the Mix tool, the execution of the script can be resumed with the Mix `resume` command. Second, if the script is started at the UNIX script driver shell prompt, the UNIX `kill` command can be used to send the `SIGRESUME` signal to the script process. See `$EMPOWER/h/empower.h` for the `SIGRESUME` value on your system. You also can resume scripts from Monitor with the "`R`" commands

If the `LCMD` (a default option) option is set, a resume message with a time stamp will be recorded in the log file. A sample of the resume message is shown below:

```
>>>     33 Resume() 13:37:30.54
```

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO

`kill(1)` in your UNIX User's Guide

FUNCTION **SysKeyDown**

SYNTAX
```
void SysKeyDown(key)
unsigned int key;
```

DESCRIPTION *Parameters*

`key` A Microsoft Windows virtual key code value

The `SysKeyDown()` function is inserted in the script file automatically during Capture to indicate that a keyboard key on the PC was depressed while the Alt key was held down. 'Sys' implies that the keystrokes are being sent to the System Menu of the current window. During script execution in Display mode, this function is used to emulate pressing the specified key down with the Alt key. In Non-Display mode, this function emulates type delay for pressing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some examples of valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

Key events may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example script segment demontrates using various key events. In this case, the user pressed the Alt key (`VK_MENU`) and then pressed and released the F key. Then the user released the Alt key and pressed the left arrow key, and pressed and released the c key:

```
CurrentWindow("Program Manager - [Empower/CS]",36,24,324,543);

Think(2.14);

SysKeyDown(VK_MENU);
SysKeyPress('F');
SysKeyUp(VK_MENU);
KeyPress(VK_LEFT);

Type("c");
```

SEE ALSO KeyDown, KeyPress, KeyUp, SysKeyPress, SysKeyUp

FUNCTION SysKeyPress

SYNTAX

```
void SysKeyPress(key)
unsigned int key;
```

DESCRIPTION

Parameters

`key`	A Microsoft Windows virtual key code value

Comments

A `SysKeyPress()` is inserted in the script file during Capture and translates into a down/up key sequence while the Alt key is held down. A `SysKeyPress()` indicates that no user events have occurred between a SysKeyDown/Up. If the key is the same for consecutive SysKeyDown and SysKeyUp events, the key events automatically will be translated into a SysKeyPress. 'Sys' implies that the keystrokes are being sent to the System Menu of the current window.

During script execution in Display mode, this function is used to emulate pressing and releasing the specified key. In Non-Display mode, this function emulates the type delay for pressing and releasing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

Key events may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES

The following example script segment demontrates using various key events. In this case, the user pressed the Alt key (`VK_MENU`) and then pressed and released the F key. Then the user released the Alt key and pressed the left arrow key, and pressed and releasing the c key:

```
CurrentWindow("Program Manager - [Empower/CS]",36,24,324,543);

Think(2.14);

SysKeyDown(VK_MENU);
SysKeyPress('F');
SysKeyUp(VK_MENU);
KeyPress(VK_LEFT);

Type("c");
```

SEE ALSO KeyDown, KeyUp, SysKeyDown, SysKeyPress, SysKeyUp

FUNCTION **SysKeyUp**

SYNTAX

```
void SysKeyUp(key)
unsigned int key;
```

DESCRIPTION

Parameters

`key` A Microsoft Windows virtual key code value

Comments

The `SysKeyUp()` function is inserted in the script file automatically during Capture and indicates that a keyboard key on the PC was released while the Alt key was held down. 'Sys' implies that the keystrokes are being sent to the System Menu of the current window. During script execution in Display mode, this function is used to emulate a releasing the specified key. In Non-Display mode, this function emulates the type delay for the key event.

The parameter `key` represents a Microsoft Windows virtual key code value. Some valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

Key events may be edited within the script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example script segment demontrates using various key events. In this case, the user pressed the Alt key (`VK_MENU`) and then pressed and released the F key. Then the user released the Alt key and pressed the left arrow key, and pressed and released the c key:

0096794.102600

```
CurrentWindow("Program Manager - [Empower/CS]",36,24,324,543);

Think(2.14);

SysKeyDown(VK_MENU);
SysKeyPress('F');
SysKeyUp(VK_MENU);
KeyPress(VK_LEFT);

Type("c");
```

SEE ALSO KeyUp, KeyPress, KeyDown, SysKeyDown, SysKeyPress

FUNCTION **Think**

SYNTAX

```
double Think(n);
double n;
```

DESCRIPTION

Parameters

n The amount of think delay measured in seconds

Comments

During script execution, `Think()` performs a delay to emulate a user's think time. The delay will be determined by the current think time distribution.

The think time distribution can be defined by any of the four think time distribution functions:

`Thinkactual()`	-	actual think time
`Thinkconstant()`	-	constant distribution
`Thinktne()`	-	truncated negative exponential distribution
`Thinkuniform()`	-	uniform distribution

`Think()` functions are inserted in the script file during Capture when the EMPOWER/CS user pauses before performing any activity. During Capture, you may specify the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script file. The `Think()` function is inserted in the captured script file only after the specified time has elapsed. The parameter of the `Think()` function specifies the amount of seconds elapsed.

Because the script was captured with a think time value, you must always have a value specified in the `Think()` functions during script execution (even if the value is zero) regardless of the think time distribution.

0565765946.102600

The functions for think time distribution can be placed anywhere in the script to emulate thinking delay of a real user based upon your testing needs.

RETURN VALUE

`Think()` returns the amount of delay time incurred.

EXAMPLES

The following example script segment demonstrates that the user paused before pressing two of the buttons during the Capture session:

```
Think(5.11);
ButtonPush("Employee Records|Next",726,439);


AppWait(2.09);
WindowRcv("SfPt");
ButtonPush("Employee Records|Next",726,439);
ButtonPush("Employee Records|Delete",714,344);

WindowRcv("SfPt");

Think(3.70);
ButtonPush("Employee Records|Close",714,157);
```

The following script segment defines the constant think time distribution to be three seconds and will perform a think delay only when `Think()` functions are executed.

```
Thinkconstant(3);
Think(1.23);
```

SEE ALSO

Seed, Thinkactual, Thinkconstant, Thinktne, Thinkuniform

FUNCTION Thinkactual

SYNTAX `Thinkactual();`

DESCRIPTION

During Capture, you may specify the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script file. The `Think()` function is inserted in the captured script file only after the specified time has elapsed. By default, if no activity is captured after two seconds, EMPOWER/CS will insert a think time function of at least two seconds and the time elapsed until the next action occurs. The parameter of the `Think()` function specifies the amount of seconds elapsed.

During script execution, `Thinkactual()` tells the script to use the actual `Think()` functions and values that were captured during your Capture session.

RETURN VALUE (not applicable)

EXAMPLES

The following example demonstrates using this function within a script. When you edit your script, insert `Thinkactual()` as shown below:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */


Typerate(5);           /* Typing delay in CPS */
Pointerrate(150);      /* Pointerrate in PPS */
Thinkactual();         /* Think delay */
Seed(getpid());        /* Seed random number generator */
Timeout(300, CONTINUE);/* What to do if function takes too long */
Dberror(CONTINUE);     /* What to do on Database errors */
Unset(NOTIFY);         /* Don't display warnings. I'll use Mon to find them */

Beginscenario("example1");
```

SEE ALSO Think, Thinkconstant, Thinktne, Thinkuniform

FUNCTION **Thinkconstant**

SYNTAX

```
void Thinkconstant(f)
double f;
```

DESCRIPTION *Parameters*

`f` The amount of thinking delay measured in seconds

Comments

`Thinkconstant()` defines a constant think time distribution. The argument `f` is a double value that specifies the amount of thinking delay in seconds.

Think delay will be performed whenever a `Think()` function is executed.

The think time distribution will be active until it is reset by another think time distribution function.

Multiple `Thinkconstant()` functions can be inserted throughout the script file when editing your script.

RETURN VALUE (not applicable)

EXAMPLES The following script segment demonstrates how to generate think times from a constant distribution of three seconds and perform think delay only when `Think()` functions are executed:

```
Thinkconstant(3);
Think(0);
```

SEE ALSO Think, Thinkactual, Thinktne, Thinkuniform

FUNCTION Thinktne

SYNTAX

```
void Thinktne(min, avg, max)
double min, avg, max;
```

DESCRIPTION

Parameters

`min`	The minimum value of the think time distribution
`avg`	The average value of the think time distribution
`max`	The maximum value of the think time distribution

Comments

`Thinktne()` defines a truncated negative exponential think time distribution. The parameters `min`, `avg`, and `max` are double values that specify the minimum, average, and maximum values of the distribution, respectively. The values are in seconds.

Think delay will be performed whenever a `Think()` function is executed.

The think time distribution will be active until it is reset by another think time distribution function. The `Seed()` function (which seeds the EMPOWER/CS random number generator) helps to determine think delays for this think time distribution.

Multiple `Thinktne()` functions can be inserted throughout the script file when editing your script.

RETURN VALUE (not applicable)

EXAMPLES The following script segment is an example of how to generate think times from a truncated negative exponential distribution. The minimum value is 2 seconds, the average value is 5 seconds, and the maximum value is 25 seconds:

```
Thinktne(2.0, 5.0, 25.0);
Think();
```

SEE ALSO Seed, Think, Thinkactual, Thinkconstant, Thinkuniform

FUNCTION Thinkuniform

SYNTAX

```
void Thinkuniform(min, max)
double min, max;
```

DESCRIPTION

Parameters

`min` The minimum value of the think time distribution

`max` The maximum value of the think time distribution

Comments

`Thinkuniform()` defines a uniform think time distribution. The parameters `min` and `max` are double values that specify the range of think delay in unit of seconds.

Think delay will be performed whenever a `Think()` function is executed.

The think time distribution will be active until it is reset by another think time distribution function. The `Seed()` function (which seeds the EMPOWER/CS random number generator) helps to determine think delays for this think time distribution.

Multiple `Thinkuniform()` functions can be inserted throughout the script file when editing your script.

RETURN VALUE (not applicable)

EXAMPLES

The following script segment is an example of how to generate think times from a uniform distribution between 30 and 45 seconds and perform think delay only when `Think()` functions are executed.

```
Thinkuniform(30, 45);
Think();
```

SEE ALSO

Seed, Think, Thinkactual, Thinktne, Thinkuniform

FUNCTION **Time**

SYNTAX

```
void Time(p)
struct timevalue *p;
```

DESCRIPTION

Parameters

`p` A structure where the current time on the UNIX driver will be stored

Comments

`Time()` obtains the current time on the UNIX script driver machine. The time will be stored in a structure pointed to by `p`. The structure is defined in `$EMPOWER/h/empowercs.h` as:

```
struct timevalue {
      long sec;   /* seconds since Jan 1. 1970 */
      short hsec; /* and hundredths of a second */
}
```

RETURN VALUE (not applicable)

SEE ALSO Eventtime, Difftime

0096794.1026004

FUNCTION **Timeout**

SYNTAX

```
int Timeout (n, f)
int n;
int (*f) ();
```

DESCRIPTION

Parameters

n An integer that specifies the elapsed time (in seconds) after which a timeout should occur

f The function to execute when a timeout occurs

Comments

During script execution, `Timeout()` specifies if and when a timeout should occur during response reading from the server. The default function `Timeout(300, CONTINUE)` is inserted at the top of every EMPOWER/CS script created. You can change this default when editing your script file.

To execute a script successfully, all events on the server must occur in the same way as they occurred during Capture. If an event occurs other than the set of expected events, the script will continue to wait for the expected events to occur. A timeout identifies expected events that do not occur.

Occasionally, script execution can not continue because of unexpected events on the server or in Display mode the client or server. The resulting unexpected responses can identify an abnormal error, loss of data between the client and server, or an unexpected screen image. If your executing script encounters a timeout, a `WindowRcv()` may need to be modified. If a timeout occurs when an emulated user attempts to connect to the SUT, you should ensure that the database on the server is available.

The function (`f` parameter) designated to handle the timeout condition can be either an EMPOWER/CS function or a user-defined function.

The two EMPOWER/CS functions are `EXIT` and `CONTINUE`. The `EXIT` function terminates the script. The `CONTINUE` function causes premature return from the executing function, and execution of the next function in the script. A user-defined function can be specified to handle a timeout if the `EXIT` and `CONTINUE` functions are not desirable.

If a timeout occurs during script execution, review the log file to determine the cause. In some cases, the expected behavior did occur, but it took longer than the current timeout delay. If this is the case, you should edit the script to increase the timeout threshold with a new `Timeout()` function.

By default, when a script executes, EMPOWER/CS does not display timeout information on the UNIX machine when a timeout occurs because the EMPOWER/CS function `Unset(NOTIFY)` is inserted in each script when it is created. If you wish to display timeout information during script execution, use the `Set(NOTIFY)` function.

RETURN VALUE

`Timeout()` returns the previous timeout value. An `ETIMEOUT` value will be returned by the executing function upon timeout.

EXAMPLES

The following is an example of how a Timeout() function may appear within a script:

```
Timeout(60, EXIT);
```

SEE ALSO

Dberror, Set, Unset

FUNCTION **Type**

SYNTAX

```
Type(str, .../* args */)
char *str, *args
```

DESCRIPTION

Parameters

`str`	A string containing the characters typed at the PC
`args`	Optional variable arguments

Comments

The `Type()` function is inserted into your script file during Capture when keystrokes are entered from the PC keyboard. The `str` parameter may include standard keyboard keys, except for those defined as MS Windows virtual key codes (such as `VK_CONTROL`, `VK_ESCAPE`, etc.) and the non-displayed key combinations defined below. Each key is translated into a KeyDown/Up pair for the emulation.

Non-displayed key combinations are shown in the parameter of the `Type()` function using the standard UNIX technique in which the character ^ represents the Control key on the keyboard and the key combination Control-M (`^M`) indicates a carriage return. Common control sequences captured into the `Type()` function are listed below:

`^M`	Carriage Return
`^H`	Backspace
`^I`	Tab
`^?`	Delete

During script execution in Display mode, the `Type()` function is used to emulate typing the specified keystrokes. In Non-Display mode, this function is used to emulate the type delay for typing the characters specified in `str`.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `Type()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments must be string pointers. The string conversion specification is provided by the characters `%s`.

RETURN VALUE (not applicable)

EXAMPLES In the following `Type()` function, the PC user typed the character `c`, then pressed the `RETURN` key. The RETURN key is indicated by the characters `^M`.

```
Type ("c^M");
```

Type strings are very easy to modify and can be edited to alter the keystrokes typed by an emulated user. For example, the following `Type()` function is entered in a query:

```
Type("13402^M");
```

It can be modified to:

```
Type("8704^M");
```

FUNCTION **Typerate**

SYNTAX

```
double Typerate(n)
double n;
```

DESCRIPTION

Parameters

n The typing speed measured in characters per second

Comments

`Typerate()` sets the emulated typing speed for a script. Typing speed is measured in characters per second. The default function `Typerate(5)` is inserted at the top of each script created by EMPOWER/CS. You may change this default when editing the script. If you specify a type rate of zero, then no delay is used during script execution. Multiple `Typerate()` functions may be used in a script to specify, for example, different type rates when using different applications.

During script execution, each time that the emulated user is supposed to be typing at the keyboard, the script will pause a length of time defined as the number of characters to be typed times the type rate specified. For example, if the emulated user must type twelve characters and the type rate is specified as six characters per second (`Typerate(6)`) then the script will pause two seconds when the emulated characters are to be "typed."

Care must be taken when inserting `Typerate()` functions into a script. The response time statistics for scenarios and functions are generated from the user's perspective. Therefore, they will take into account the time it takes for the emulated user to enter information at the keyboard. This is particularly important when you are comparing the performance of similar scripts. Any `Typerate()` functions inserted into one script must be inserted in the same locations in the other script(s) to provide a meaningful comparison.

If you specify rate less than zero, script will exit, receive an error message

RETURN VALUE This function returns the old typerate value

EXAMPLES In the following script segment, the type rate delay is set at 6 characters per second:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */


Typerate(5);            /* Typing delay in CPS */
Pointerrate(150);      /* Pointerrate in PPS */
Thinkuniform(1,2.5);   /* Think delay */
Seed(getpid());        /* Seed random number generator */
Timeout(300, CONTINUE);/* What to do if function takes too long */
Dberror(CONTINUE);     /* What to do on Database errors */
Unset(NOTIFY);         /* Don't display warnings.  I'll use Mon to find them */

Beginscenario("foo");
```

SEE ALSO Pointerrate

FUNCTION Unset

SYNTAX

```
long Unset(n)
long n;
```

DESCRIPTION

Parameters

`n` The EMPOWER/CS option to be turned off

Comments

`Unset()` turns off EMPOWER/CS options during script execution. Valid options are listed below:

OPTION	DESCRIPTION
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in a `WindowRcv()`
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	enable logging

Default options are `LCMD` and `LOGGING`.

RETURN VALUE `Unset()` returns a long value containing the previously set options.

EXAMPLES The following examples demonstrate using `Unset()`.

```
Unset(FLUSH);
Unset(LCMD);
```

SEE ALSO Set

FUNCTION **Username**

SYNTAX

```
Username(lognum, username)
int lognum;
char *username;
```

DESCRIPTION

Parameters

`lognum` An identifier of a logon communication structure

`username` A user name specified for logging on the SUT

Comments

The `Username()` function is inserted into your script when a user name was entered to logon to the SUT. This function will occur before a `Logon()` function in your script.

During script execution, `Username()` sets the name of a user logging on to the database before a `Logon()` function executes.

RETURN VALUE

If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES

In the following example, the user `scott` attempted to logon to the SUT:

```
Type("scott^Itiger^IFOO^M")

Username(LOG1, "scott@FOO");
Password(LOG1, "tiger");
Logon(ORACLE1, LOG1);
```

SEE ALSO

AppName, Hostname, Language, Password, Servername

FUNCTION **WindowRcv**

SYNTAX

```
void WindowRcv(pattern)
char *pattern;
```

DESCRIPTION

Parameters

`pattern` Command pattern for drawing windows on screen

Comments

The `WindowRcv()` function is inserted into the script file during Capture when an activated window is drawn on screen and usually follows an `AppWait()` call. Consecutive `WindowRcv()` functions may be inserted into the script during Capture. This function is used during script execution in Display mode to ensure that the activated window is drawn on screen. In Non-Display mode, this function is ignored because the `AppWait()` function simply emulates the time taken to draw the window.

The parameters of this function are MS Windows standard two-letter pneumonics. The parameters can be any of the following commands:

Command	Description
`Cw`	Create Window
`Pt`	Paint
`Dw`	Destroy Window
`Co`	Command
`Sf`	Set focus
`Ac`	Activate
`Sz`	System Command

RETURN VALUE (not applicable)

EXAMPLES

The following example demonstrates typical `WindowRcv()` commands from a script file:

```
AppWait(0.05);
WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwCwSfAcSzPt");
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
CurrentWindow("Run",429,491,880,764);
```

SEE ALSO

AppWait

3.0 Error Messages

The following pages contain a list of warning and error messages that may be generated by EMPOWER/CS.

Most of the warning messages generated by EMPOWER/CS will be preceded with the word `warning` and most error messages will be preceded with the word `error`.

Usually, names of EMPOWER/CS tools and script names or IDs precede the warning and error messages. This feature helps you to locate the tool and script generating the warning or error message.

All warning and error messages are listed in alphabetical order followed by detailed descriptions and corrective actions.

`All input files do not have the same unit of time`

The input files to Draw do not contain the same unit on the line identifying the duration of the reports. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

`Arithmetic operation invalid for type string`

An arithmetic operation (e.g., `Gv_add()`) was attempted on a character string type global variable. Only read and write operations are valid for character string variables.

`Attempt to obtain the time of an unknown event`

`Eventtime()` was passed an invalid argument. Check script for mistyping. Make sure that you call `Eventtime()` with an argument that is one of `TBF`, `TEF`, `TBS`, or `TES`.

`Bad within value`

A `-w` argument to Report was followed by an invalid value. The value must specify a time in ss, mm:ss, hh:mm:ss format. The time can optionally be followed by a decimal point and an integer to identify a fraction of a second. Check the syntax of the Report command and rerun.

`Bars are not placed proportionally`

Draw was unable to separate the bars in a chart with space that is proportional to the number of users represented by each bar. Select a different set of input files with different numbers of users if you must have bars that are proportionally spaced.

`Bitwise operation invalid for type double`
`Bitwise operation invalid for type float`
`Bitwise operation invalid for type string`

A bit-wise operation (e.g., `Gv_and()`) was attempted on a double, float, or character string type global variable. Only read, write, and arithmetic operations are valid for

double and float variables. Only read and write operations are valid for character string variables.

`Bitwise relation invalid for type double`
`Bitwise relation invalid for type float`
`Bitwise relation invalid for type string`
A bit-wise relation (e.g., "|", "&") was attempted on a double, float, or character string type global variable. Only logical comparisons are valid for these variable types.

`Can't accept duplicate characters. Try again`
Draw requested `LEGEND` characters in interactive mode and you repeated one or more characters. Your characters must be unique. Enter a different set of characters. Enter quit to exit Draw.

`Can't accept duplicate selection. Try again`
Draw requested a set of choices in interactive mode and you repeated one or more of the choices. Your choices must be unique. Enter a different set of choices. Enter quit to exit Draw.

`Can't allocate global variable:  different type already in use`
The global variable specified exists but is a different type than specified in the `Gv_alloc()` statement. The "`type`" argument of `Gv_alloc()` must match the existing variable type.

`Can't attach to shared memory segment`
`Can't change size of existing shared memory segment`
A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't continue. Draw exited`

`Can't continue. Specification generated is stored in [specification]`

`Can't continue. Specifications generated are stored in [specification]`

Draw encountered a condition that prevented it from continuing. Correct the condition. Specifications created prior to the error condition are saved in the file specified.

`Can't create control semaphore`

`Can't create shared memory segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't create global variable: maximum exceeded`

An attempt was made to create an additional global variable beyond the limit allowed by default. If more global variables are required, remove all existing variables with the `gv_seg -r` command which will destroy all global variables, then, with `gv_seg newlimit` where `newlimit` is a new number, create a new shared segment with a higher limit. Use `gv_stat -s` to determine the current limit.

`Can't create shared memory segment`

`Can't detach from global variable segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't detach from global variable segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't exec [cc statement]`

Cscc is trying to execute the C compiler on your UNIX script driver. The C compiler is not in your `PATH` environment variable or does not have execute permission. Ensure that the C compiler can be executed at the command line.

`Can't find number of users in [input]`

An input file to Draw does not contain a line identifying the number of users in the report. Make sure the input file is a `.STD` file. Do not edit the `.STD` files created by Report.

`Can't find the specified event in all of the input files`

A Draw specification identifies an event that does not exist in all of the input `.STD` files. An event is the name of a scenario or function. The event must exist in all input files. Correct the specification to identify an existing event. Check spelling. Make sure your `INPUT` variable identifies the correct input files.

`Can't find the specified event type in the input file`

A Draw specification identifies an event that does not exist in the input `.STD` file. An event is the name of a scenario, function or transaction. Correct the specification to identify an existing event. Check spelling. Make sure your `INPUT` variable identifies the correct input file.

`Can't find the specified field in the input file`

A Draw specification identifies a field that does not exist in the input `.STD` file. Check spelling in the specification. Make sure your `INPUT` variable identifies the correct input file.

`Can't find the specified field in all of the input files`

A Draw specification identifies a field that does not exist in all of the input `.STD` files. Check spelling in the specification. Make sure your `INPUT` variable identifies the correct input files.

`Can't find unit of time in [input]`

An input file to Draw is missing the line containing the duration of a report. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

`Can't find [file]`

Cscc can not find a function library or header file required for compilation. Make sure your `EMPOWER` environment variable is set to the location of the installed EMPOWER/CS software. Make sure the `EMPOWER` environment variable is exported.

`Can't fork`

Make sure the UNIX script driver kernel is configured with adequate resources to execute processes. Check the UNIX script driver console for kernel error messages. Typical resources that need increasing are the maximum number of processes in a system and the maximum number of processes per user.

`Can't get control semaphore`
`Can't get shared memory segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs`

shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't open /dev/null`

An attempt to open `/dev/null` failed. Make sure `/dev/null` exists on your UNIX script driver and has read and write permission. Make sure `/dev/null` is a special character device.

`Can't open current directory`

The name of the current directory can not be obtained. A call to `getcwd()` failed. Make sure the current directory has read permission.

`Can't open [log] for writing`

Log specified can't be opened. Check syntax of command to execute your script. Check script table entries if you are executing scripts with Mix. Remember that a port must be specified to specify a log. Make sure the directory in which the log is being written has write permission for you. Make sure an existing copy of the log has write permission for you.

```
Can't open [input]
Can't open [output]
Can't open [mixlog]
Can't open [cscc*.o]
Can't open [script table]
Can't open [script.c]
Can't open [specification]
```

An attempt to open a file failed. If the file is to be created, make sure you have write permission to the directory in which the file will be placed and write permission to an existing version of the file if the file exists. Make sure the file system in which the file is to be created is not full. If the file is to be read, make sure the file exists and that you have read permission on the file.

`Can't read [file]`

Cscc can not read a function library or header file required for compilation. Make sure that you have read permission on the EMPOWER/CS libraries and header files.

`Can't rebind stdin of script`

EMPOWER/CS rebinds the standard input file descriptors of a script running in the background to prevent the script from receiving interrupt signals generated at the keyboard. A `dup()` function failed. The `dup()` was interrupted by a signal or reached a maximum number of open files. Make sure there is an adequate number of open files permitted on the UNIX script driver machine and that the maximum number of open files per user is large.

`Can't re-initialize: global variable active or protected by other`

The `gv_init` command was entered to reset a variable which is in use or protected. Check for active scripts using the variable. If no scripts are running which use the variable, remove the variable before attempting the `gv_init` command again.

`Can't remove allocated variable`

The `gv_rm` command was used while the global variable was allocated to at least one script. Wait for scripts to finish or use the `-f` option.

`Can't remove shared memory segment`
`Can't remove shared semaphore segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't start [scriptid]`

`Can't start [scriptid.n]`

A script identified in a script table can't be executed. The script is probably misspelled, not in your `PATH` environment variable, or does not have execute permission. Check syntax and spelling in the script table. Ensure that the script can be executed at the command line.

Make sure the UNIX script driver kernel is configured with adequate resources to execute processes. Check the UNIX script driver console for kernel error messages. Typical resources that need increasing are the maximum number of processes in a system and the maximum number of processes per user.

`Can't stat shared memory segment`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Can't unprotect global variable without protecting first`

A script executed the `Gv_unprotect()` function when the specified variable had not been protected by the script.

`Can't wait on self-protected global variable`

A script executed the `Gv_waitwhile()` or `Gv_waituntil()` function on a variable which the script has protected with `Gv_protect()`. A protected variable can not be updated by other scripts, so a `Gv_waitwhile()` or `Gv_waituntil()` function probably will cause an indefinite delay.

`Character = not found; ignoring the line`

Draw encountered a keyword in a specification that was not followed by an = character. Check the specification.

`Character [c] not found; ignoring the line`

Draw encountered a statement in a specification that did not contain a required character. Check the specification.

`Character string in the specification is too long`

The name of an event in an input file to Draw is too long. The maximum is 15 characters. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

`Characters should be separated with spaces`

The `LEGEND` characters in a specification to Draw are not separated by spaces. Check the syntax of the `LEGEND` statement in the specification.

`COMMENT too long.  Ignored`

Draw requested a `COMMENT` in interactive mode and your response identified one that is too long. Enter a shorter `COMMENT`. Maximum `COMMENT` is 60 characters. Enter quit to exit Draw.

`Current directory does not contain any ..STD files`

Draw can not find any files with the `..STD` suffix in the current directory. Make sure you are in the directory containing your `.STD` files.

`Division by zero undefined`

The value zero was passed as the divisor argument to a `Gv_div()` or `Gv_mod()` function.

`Empty line. Try again`

Draw is requesting input in interactive mode and none was entered. Enter a response or type quit to exit Draw.

`Error in [file]`

A specification or input file to Draw contains an error. A specific error message follows this error. Refer to the description of the specific error message for more information.

`Error near line [n] => [data]`

A specification or input file to Draw contains an error. A specific error message follows the => symbol. Refer to the description of the specific error message for more information. The line number n is often not correct.

`EVENT value is not defined properly`

A specification to Draw contains and invalid EVENT statement. Check syntax in the specification.

`Failed set all semctl`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Forced to use HIDDEN format`

A Draw specification requested bars to be drawn in CLUSTERED format. Draw was unable to fit all of the bars on the chart. Draw changed format to HIDDEN. Reduce the number of bars to be drawn if you want the chart in CLUSTERED format.

`Forcing protection of global variable protected by other`

The `gv_protect` command was used with the `-f` option while the global variable was protected by a script.

`Forcing unprotection of global variable protected by other`
The `gv_unprotect` command was used with the `-f` option while the global variable was protected by a script.

`Format incorrect. Try again`
Draw requested input in interactive mode and your response was entered in an unacceptable format. Correct the format and retry. Enter a response or type quit to exit Draw.

`Global variable already allocated`
A script attempted to allocate access to a variable which has already been allocated to the script. Each variable may be allocated only once in each script unless it has been de-allocated with `Gv_free()`.

`Global variable already protected`
A script attempted to protect a variable that is already protected. The variable must be unprotected before it can be protected again.

`Global variable does not exist`
A specified variable was not created with the `gv_init` command, or was removed with the `gv_rm` command.

`Global variable name allocation problem`
Generally, this error is caused when a `fork()` system call has been executed after a variable was allocated to the script. If this error occurs, you can not `fork()` after allocating variables.

`Global variable not currently allocated`

A Global Variable function specified a variable that is not currently allocated to the script. You must allocate a variable with the `Gv_alloc()` function before using the variable.

`Global variable protected by other process`

The `gv_protect` command was executed to protect a variable which is currently protected by a script.

`Illegal global variable check type`

An illegal check type was specified. Use only the commands and functions specified in this reference manual.

`Illegal global variable relation`
`Illegal global variable relation length`
`Invalid global variable relation`

An illegal relation string was used with the `Gv_test()`, `Gv_waitwhile()`, or `Gv_waituntil()` function.

`Illegal return value from semop/ctl call`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

`Incomplete specification -- required fields undefined`

Draw encountered a specification that did not contain the required statements `INPUT`, `X`, `Y` and `EVENT`. Correct the specification.

`Incomplete statement in the specification. Defaults will apply`

Draw encountered a specification that did not contain a properly formatted statement. Correct the specification.

`INPUT file does not contain proper data`

`INPUT files do not contain proper data`

One of the input files to Draw contains erroneous data. Make sure you specify `.STD` files created by Report as input. Do not edit the `.STD` files that are to be used by Draw.

`Input file has too many columns`

Draw encountered an input file that had more that 20 columns of possible `Y` values. The maximum in a `.STD` file is 20. Create the `.STD` files with fewer `Y` values and retry.

`INPUT was redefined; the last one will override`

More than one `INPUT` statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

`Invalid character in name of function`

A `Beginfunction()` or `Endfunction()` function contained unprintable characters. Check the argument of the failing function. The argument must be a string.

`Invalid character in name of scenario`

A `Beginscenario()` or `Endscenario()` function contained unprintable characters. Check the argument of the failing function. The argument must be a string.

`Invalid choice. Try again`

Draw requested input in interactive mode and your response was not one of the possible choices. Enter a choice from the list displayed by Draw. Enter quit to exit Draw.

`Invalid global variable name length`

The name of the variable is either too long or was not specified. The maximum name length is 14 characters.

`Invalid global variable operation`

An invalid operation was specified. Use only the commands and functions specified in Section 7 EMPOWER/GV of the Multi-User Testing manual.

`Invalid global variable type`

An invalid variable type was specified. The type specification must match one of the valid variable types specified in Section 7 EMPOWER/GV of the Multi-User Testing manual.

`Invalid number of characters to seek`

A `Seek()` function was called with an invalid argument. The argument must be an integer or floating point greater than or equal to zero. Check the argument of the failing function.

`Invalid specification -- BEGIN statement expected`

Draw encountered a specification that did not begin with a `BEGIN` statement. Check the specification. Make sure you are identifying the correct specification.

`Invalid timeout value`

A `Timeout()` function was called with an invalid argument. The argument must be an integer or floating point greater than or equal to zero. Check the argument of the failing function.

`Invalid values in think time distribution`

A `Thinkconstant()`, `Thinkuniform()`, or `Thinktne()` function was called with one or more invalid arguments. The arguments to one function must be ascending in value since they specify minimum, average and maximum values in sequence. Remember that `Thinkconstant()` requires one argument, `Thinkuniform()` requires two arguments, and `Thinktne()` requires three arguments. Check the arguments of the failing function.

`LEGEND was redefined; the last one will override`

More than one LEGEND statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

`Missing quote`

A line in the script table does not contain an even number of " characters. The " is used when specifying arguments to scripts in the script table. Check the syntax of your script table.

`Missing script id`

A line in the script table does not contain a script ID. Make sure you have identified the proper script table on the Mix `use` command. Check syntax of lines in the script table. Remember that comments in the script table begin with a # character.

`Modulo operation invalid for type double`
`Modulo operation invalid for type float`

Modulo arithmetic is not permitted for double or float type variables.

`Must specify m before w`

The `-m` option of Report must occur before the `-w` option. Check the syntax of the Report command and rerun.

`N/A values in INPUT file`

Draw encountered an input file than contained the `N/A` symbol for a data value that you requested. The `N/A` is generated by Report if the event never occurred during the emulation. Rerun the emulation and make sure that the event requested occurs at least once or select a different event to chart.

`Negative thinktime value`

A `Thinkconstant()`, `Thinkuniform()`, or `Thinktne()` function was called with one or more invalid arguments. The arguments must be integers or floating points greater than or equal to zero. Remember that `Thinkconstant()` requires one argument, `Thinkuniform()` requires two arguments, and `Thinktne()` requires three arguments. Check the arguments of the failing function.

`No help available. recompile without -h option.`

A script was compiled with the `-h` option of Scc and subsequently executed with an invalid argument. The help information regarding the usage of arguments was not compiled in the script. Check the syntax of the command used to start the script or compile the script without the `-h` option to Scc and rerun.

`No more comments allowed`

The maximum number of comments in a specification to Draw was exceeded. The maximum number is 50. Reduce the number of comments.

`Not enough room for CLUSTERED format`

A Draw specification requested bars to be drawn in `CLUSTERED` format. Draw was unable to fit all of the bars on the chart. Draw changed format to `HIDDEN`. Reduce the number of bars to be drawn if you want the chart in `CLUSTERED` format.

`Number incorrect. Try again`

Draw requested input in interactive mode and your response was not one of the possible choices. Enter a choice from the list displayed by Draw. Enter `quit` to exit Draw.

`Number of LEGEND characters incorrect.`

The `LEGEND` characters in a specification to Draw are not separated by spaces. Check the syntax of the `LEGEND` statement in the specification. You should specify one `LEGEND` character for each `Y` value specified.

`Number of users not distributed well`

Draw was unable to separate the bars in a chart with space that is proportional to the number of users represented by each bar. Select a different set of input files with different numbers of users if you must have bars that are proportionally spaced.

`ORGANIZE value not defined properly. Default of CLUSTERED will apply`

Draw encountered a specification that did not contain an `ORGANIZE` statement that was properly formatted. Correct the specification.

`ORGANIZE was redefined, the last one will override`

More than one `ORGANIZE` statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

`Out of memory`

The UNIX script driver machine is out of virtual memory.

`Percentile out of range`

A `-p` argument to Report was followed by an invalid value. The value must be an integer greater than zero and less than or equal to one hundred. Check the syntax of the Report command and rerun.

`Performance measures in [input] not consistent`

The input files to Draw do not contain the same set of Y values generated by Report. Make sure that you use the same set of `-p` and `-w` options to Report when creating each of the `.STD` files used as input to Draw. Make sure your `INPUT` variable identifies the correct input files.

`Reached another BEGIN before keyword END`

Draw encounted a `BEGIN` statement in a specification before an `END` statement terminated the current specification. Check the specification. Make sure you are identifying the correct specification.

`Reached end of file before keyword END`

Draw encounted the end of a specification before an `END` statement terminated the current specification. Check the specification. Make sure you are identifying the correct specification.

`Received signal n`

EMPOWER/CS received a signal that it was not expecting. The signal was probably generated by pressing the interrupt key, e.g. `^C`, a `kill` command or after detecting the disconnection of a terminal. Prevent the signal from recurring and rerun. You can locate the meaning of the signal in `/usr/include/sys/signal.h`.

`Removing allocated variable`

The `gv_rm` command was used with the `-f` option while the variable was allocated to at least one script.

`Resumed at [99:99:99.99]`

Execution of a script is continuing following a `Suspend()` function. The script was resumed from Mix, Monitor, or with a `kill` command.

`Semctl call failed`

`Semop/ctl failed`

`Semop/ctl call failed`

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command and increase the number of semaphore undo structures (`Sem undo` or `sem nu`–These structures are UNIX kernel parameters). Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` to destroy the global variables and then recreating them with `gv_init` may be necessary.

`Source [script.c] newer than binary. execution continues.`

A script detected that the source version of the script has a later modification date than the binary version. The source version may have been changed and requires a recompilation. Recompile the source version of the script or move the script to another directory to eliminate the warning.

The script looks in the current directory for the source version. The name of the source version is formed by adding a .c to the name of the binary.

`Suspended at [99:99:99.99]`

Execution of a script is suspended because of execution of the `Suspend()` function or Monitor. Script execution will pause until the script is resumed.

`Timeout occurred at [99:99:99.99]. execution continues.`

The timeout condition in a script was set to CONTINUE. A timeout occurred while the script was waiting for a response from the SUT. The expected response from the SUT was not received. You should examine the log created by the script and determine whether or not the response was valid.

`Timeout occurred at [99:99:99.99]. execution terminated.`

The timeout condition in a script was set to EXIT. A timeout occurred while the script was waiting for a response from the SUT. The expected response from the SUT was not received. You should examine the log created by the script and determine whether or not the response was valid.

`TITLE must contain 60 or fewer characters`

A TITLE in a specification to Draw is too long. The maximum is 60 characters. Reduce the number of characters in the TITLE.

`Too few values. Both YMIN and YMAX are required`

Draw requested a YMIN and YMAX in interactive mode and your response did not specify both. Enter both values in integer or floating point format. Enter quit to exit Draw.

`Too many arguments in cc statement`

The maximum number of arguments that can be passed to a cc statement by Cscc is 99. The limit has been exceeded. Decrease the number of arguments by modifying your E_CFLAGS and E_LIBS environment variables.

`Too many arguments specified -- extras were ignored`

Draw encountered a statement in a specification that contained too many arguments after a = character. Check the syntax of the specification.

`Too many comments.. Ignored the previous one`

The maximum number of comments in a specification to Draw was exceeded. The maximum number is 50. Reduce the number of comments.

`Too many fields. Only one field required`

Draw requested input in interactive mode and your response identified too many choices. Enter only one choice from the list displayed by Draw. Enter `quit` to exit Draw.

`Too many fields. Select up to 3 fields. Try again`

Draw requested input in interactive mode and your response identified too many choices. Enter only up to three choices from the list displayed by Draw. Enter `quit` to exit Draw.

`Too many files. Select up to 59 files. Try again`

Draw requested input in interactive mode and your response identified too many choices. Enter only up to 59 choices from the list displayed by Draw. Enter `quit` to exit Draw.

`Too many INPUT files -- extras were ignored`

A specification to Draw identified too many `INPUT` files. The maximum number of input files is 59. Reduce the number in the specification.

`Too many LEGEND characters -- extras were ignored`

A specification to Draw identified too many `LEGEND` characters. There should be one `LEGEND` character for each `Y` value. Reduce the number in the specification.

`Too many ORGANIZE values -- extras were ignored`

A specification to Draw identified too many `ORGANIZE` values. There should be only one `ORGANIZE` value. Reduce the number in the specification.

`Too many processes`

Maximum number of processes per user on the UNIX script driver machine has been reached. Make sure there is an adequate number of processes permitted on the UNIX script driver machine and that the maximum number of processes per user is large. Reconfigure the UNIX kernel on the UNIX script driver and rerun.

`Too many values. Only YMIN, YMAX required`

Draw requested a `YMIN` and `YMAX` in interactive mode and your response specified more than the two values. Enter two values in integer or floating point format. Enter `quit` to exit Draw.

`Too many warnings -- additional warnings will not be displayed`

Draw generated too many warnings during a single execution. Correct the conditions that are causing the warnings.

`Too many within values`

More than 100 within values were specified on one command line to start Report. Reduce the number of within values or check the syntax of the Report command and rerun.

`Too many X values -- extras were ignored`

A specification to Draw identified too many `X` values. There may be up to 59 `X` values. Reduce the number in the specification.

`Too many Y values -- extras were ignored`

A specification to Draw identified too many `Y` values. There may be up to three `Y` values. Reduce the number in the specification.

`Unexpected suspend request`

A global variables shell command received a signal from a `kill` command or process. Someone else may be trying to kill your process.

`Unit of time inconsistent. Try again`

The input files selected Draw do not contain the same unit on the line identifying the duration of the reports. Select a different set of `INPUT` files from the list displayed by Draw. Do not edit the `.STD` files created by Report.

`Unit of time inconsistent in [input]`

The input files to Draw do not contain the same unit on the line identifying the duration of the reports. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

`Unrecognized keyword -- line ignored`

Draw encountered a statement in a specification that did not begin with a valid keyword. Check the specification. Make sure the keywords are followed by a space and a = character.

`Value conversion failed`

The value passed could not be converted to the variable type.

`Value in the input file is missing -- N/A assumed`

Draw encountered an input file than did not contain a `Y` value for an event that you requested. Rerun the emulation and make sure that the event requested occurs at least once or select a different event to chart. Do not edit the `.STD` files created by Report.

`Value string exceeds maximum length of a global variable string`
A string argument was longer than the maximum allowed for global variable strings. Global variable strings must be no longer than 32 characters.

`Within value out of range`
A `-w` argument to Report was followed by an invalid value. The value must be greater than zero. Check the syntax of the Report command and rerun.

`X axis TITLE must contain 30 or fewer characters`
Draw requested an `XTITLE` in interactive mode and your response identified one that is too long. Enter a shorter `XTITLE`. Enter `quit` to exit Draw.

`X was redefined; the last one will override`
More than one `X` statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

`Y axis TITLE must contain 30 or fewer characters`
Draw requested a `YTITLE` in interactive mode and your response identified one that is too long. Enter a shorter `YTITLE`. Enter `quit` to exit Draw.

`Y values are not consistent`
The `Y` values specified to Draw do not have the same unit of measure. For example, you are trying to place average response time and throughput on the same chart. Select a different set of `Y` values.

`Y was redefined; the last one will override`
More than one `Y` statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

[This page intentionally left blank]

4.0 EMPOWER/CS Technical Support

Contact PERFORMIX Technical Support if you experience problems with any aspect of EMPOWER/CS. For problems with script development, script execution, reporting, or other operational aspects of EMPOWER/CS, please have the following information at hand when you contact us:

- Any relevant script files
- Any relevant log files
- The Mix table
- The Mix log file
- Your software Version Number
- Your software Serial Number

To determine your software version and serial numbers, use any EMPOWER/CS tool to display the copyright banner. For example:

```
$ cscc  -
Cscc:   EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Usage:
      . . . .
```

EMPOWER/CS software is identified with a three-digit version number, such as "1.0.1". The first digit represents the major version cycle, the second digit represents the minor version cycle, and the third digit represents the patch level. This patch level increases with bug fixes.

The serial number displayed in the copyright banner has two elements. The first portion identifies the five-digit software license number and is used by PERFORMIX support personnel to identify your particular platform, restrictions, and terms and conditions. The second part of the serial number identifies the copy number. The letter preceding the serial number identifies the type of license (i.e., "R" for regular, "S" for site, etc.).

When calling for technical support, you will need to give both the three digit version number and the entire serial number.

You may contact the PERFORMIX main office at (703) 448-6606 between 9:00 am and 5:00 pm Eastern time.

For your convenience, you also may request help by fax at any time of day or night. The PERFORMIX main office fax number is (703) 893-1939. In your fax, include the software version and serial numbers, a thorough description of the problem, and any supportive data (e.g., scripts, error messages, etc.).

You also may email relevant information to support@performix.com.